Lighting and Shading

CS 355: Introduction to Graphics and Image Processing

Kinds of Lighting

- Direct:
  Light falling on an object directly from a light source

- Indirect: ← More in CS 455
  Light falling on an object after being reflected off
  (or going through) other objects

- Ambient:
  General light bouncing around and scattered enough to be
  effectively “everywhere”

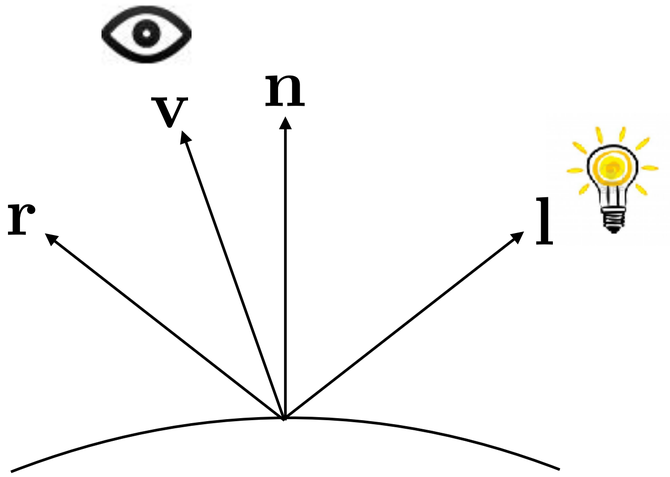
Light Sources

We'll focus on this for now

- Point
- Area
- Spot
- and many other models...

Basic Geometry of Lighting

- The surface **n**ormal
- The **l**ighting direction
  (to the light)
- The **v**iewing direction
  (to the eye/camera)
- The **r**eflected light direction



Surface Reflectance

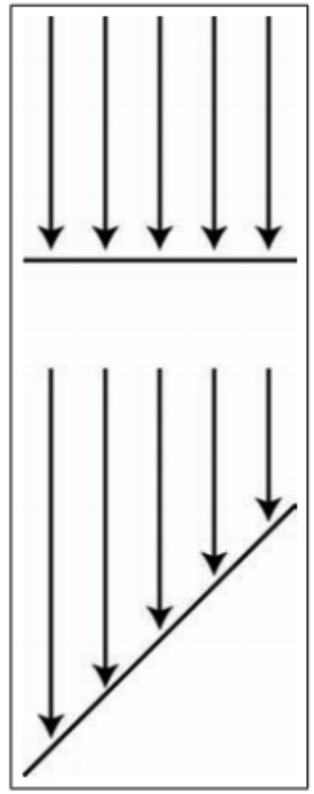
- Most objects don't give off light
  - reflect some of the light that falls on them
  - absorb the rest
- The wavelengths reflected give the object its color
- The effect is multiplicative: i.e., "reflects 40% of the green light"
- If we model the light as RGB, we can also model reflectance as RGB
- Reflectance is also sometimes called *albedo*

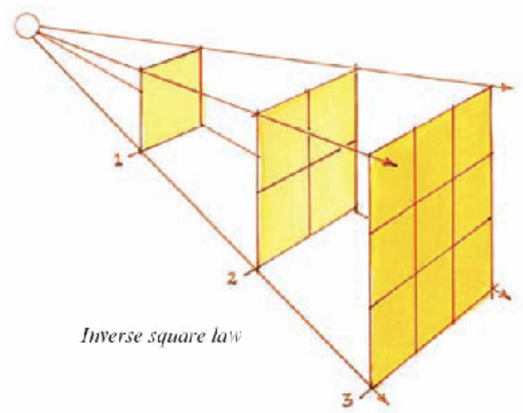
Irradiance

- We sometimes say “the amount of light”
- But it’s really how much light *per unit area*
- This quantity is called the *irradiance*

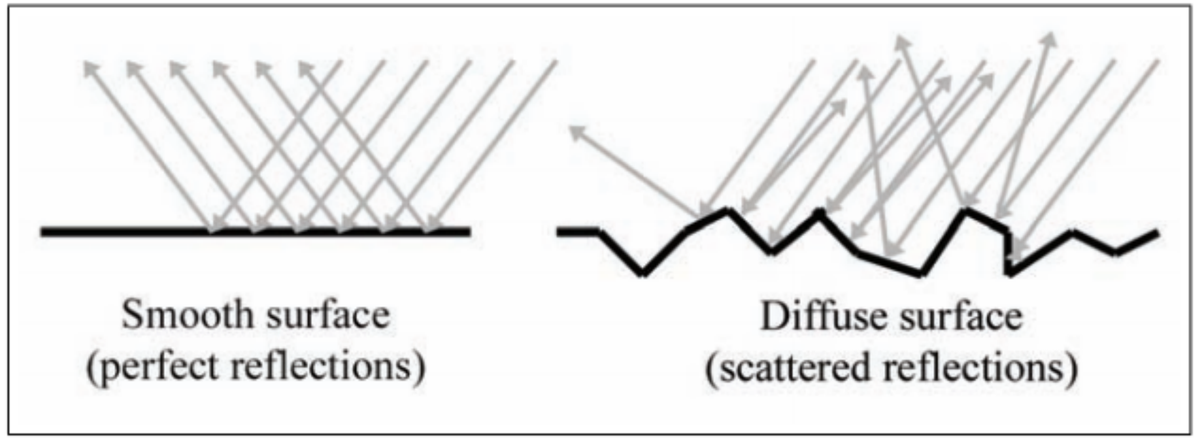
Irradiance

- Two important properties:
  - Irradiance falls of with the square of the distance
  - Irradiance is less when falling on a slanted surface





Specular vs. Diffuse

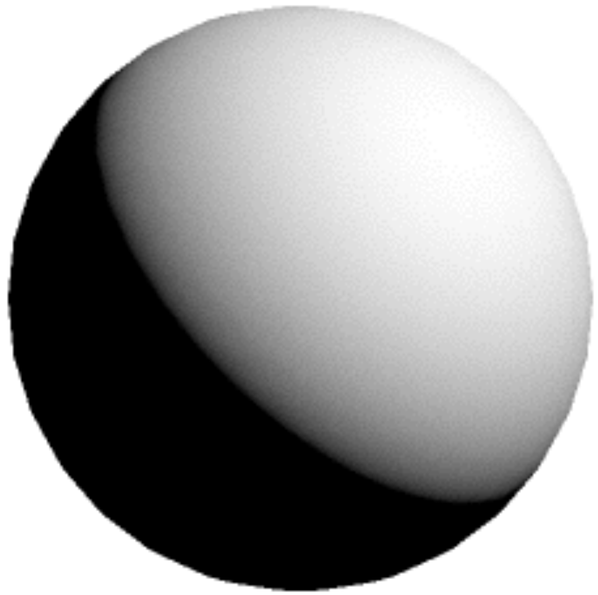


Some light is reflected perfectly (specular)

Some light is scattered (diffuse)

Diffuse Reflection

- Light scattered in every direction is called the *diffuse* part of the reflected light
- A perfectly diffuse surface is called *Lambertian*
- Only lighting direction matters
- Viewing direction *does not*



A Simple Diffuse Model

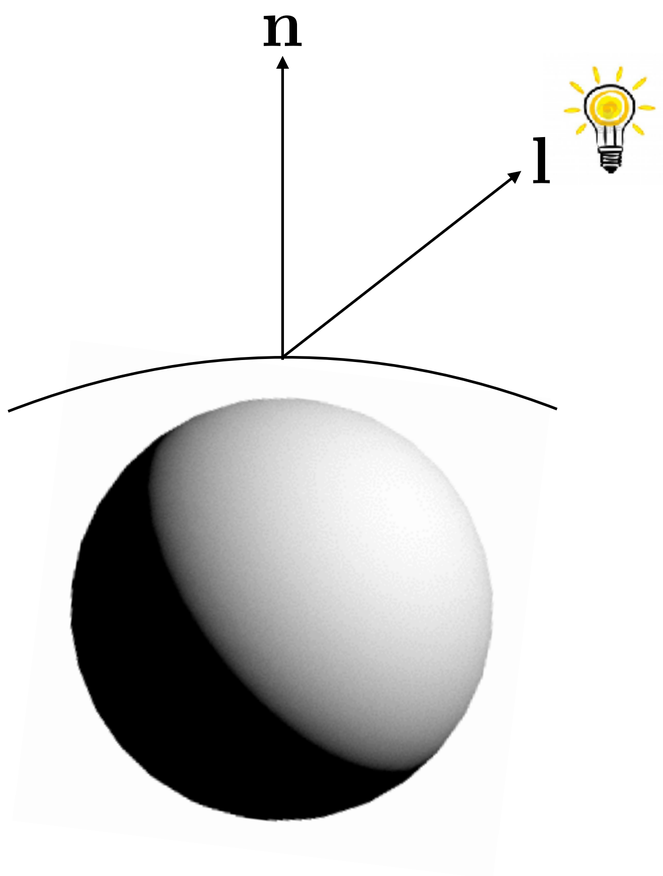
pointwise multiply RGB

$$\mathbf{c}_{\text{diff}} = (\mathbf{s} \otimes \mathbf{m}_{\text{diff}})(\mathbf{n} \cdot \mathbf{l})$$

diffuse reflected color

source intensity

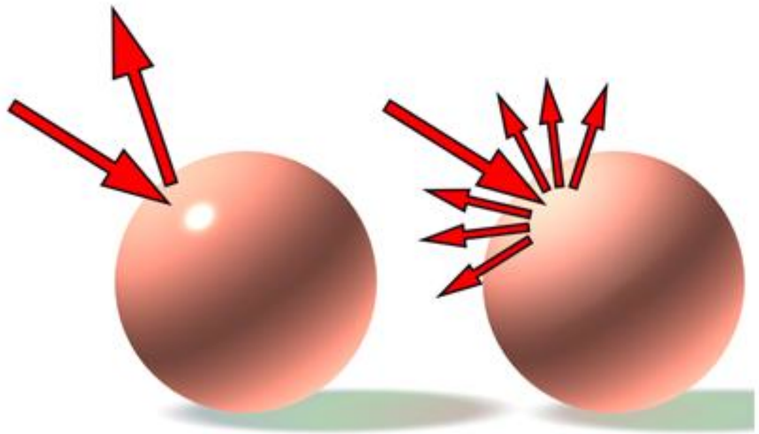
material diffuse reflectance

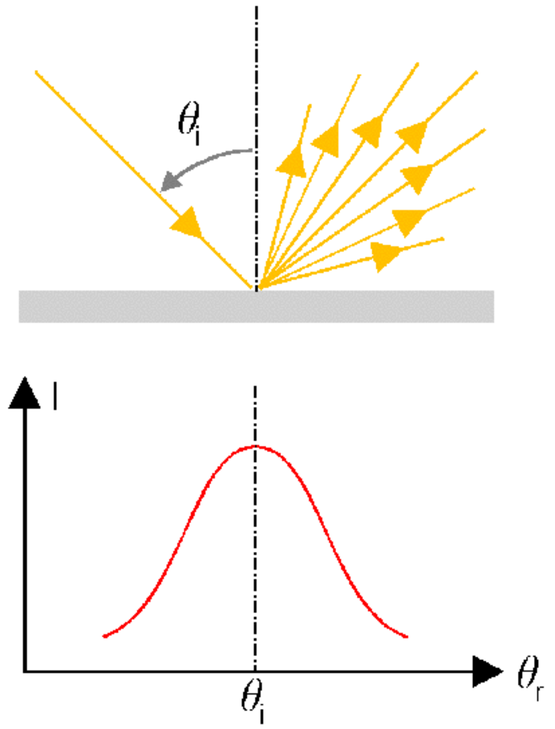


Assumes constant lighting direction and strength

Specular Reflections

Specular Reflections





Angle of reflection = Angle of incidence
(but may be blurred)

Specular Reflections

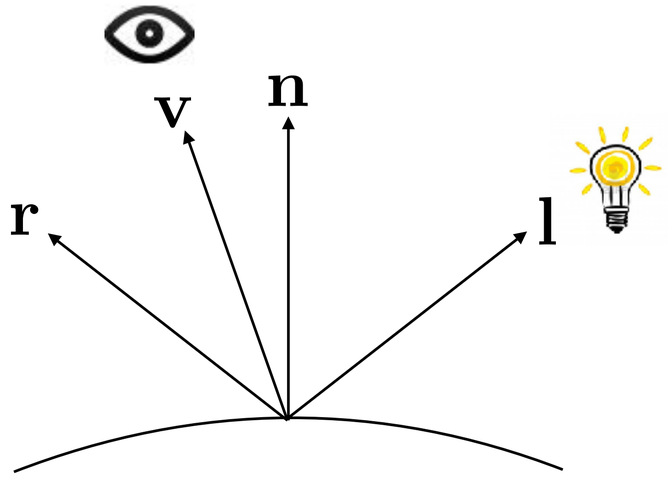
how glossy the surface is

$$\mathbf{c}_{\text{spec}} = (\mathbf{s} \otimes \mathbf{m}_{\text{spec}})(\mathbf{v} \cdot \mathbf{r})^{m_{\text{gls}}}$$

specular reflected color

source intensity

material specular reflectance



Specular Reflections

m_{gls}

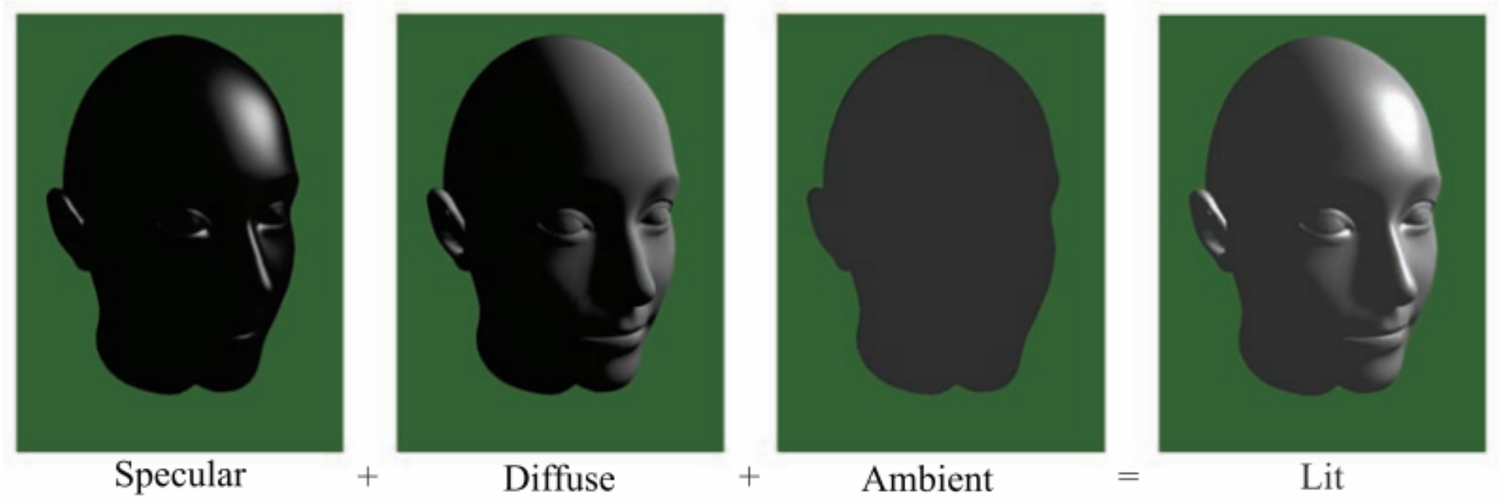
$\mathbf{m}_{\text{spec}}$

Ambient Reflection

- Ambient light is “all around”,
  so directions don’t matter
- Just the product of the ambient light
  and the surface reflectance

$$\mathbf{c}_{\text{amb}} = \mathbf{s}_{\text{amb}} \otimes \mathbf{m}_{\text{amb}}$$

All Together Now…



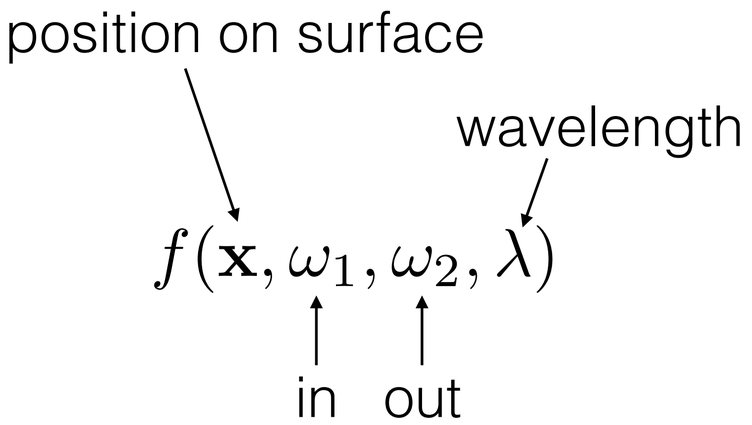
This is called the *Phong* model
(the *Blinn* model is similar with slightly different specular)

BRDFs

- The Phong model is only an approximation
- Real reflections are not a simple mix of pure diffuse and pure specular
- Function of both incoming direction and outgoing direction
- Reflectance isn’t constant across the surface

BRDFs

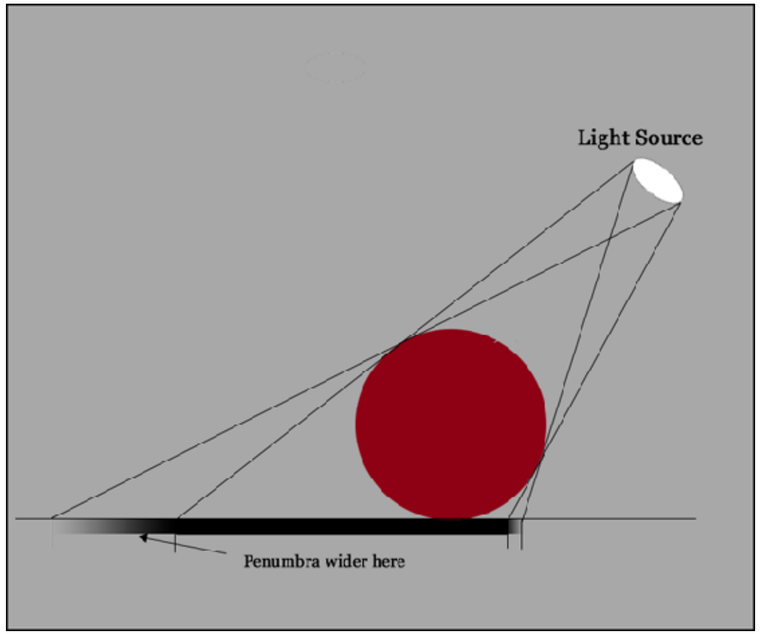
- The Phong model is only an approximation
- Not a simple mix of pure diffuse and pure specular
- Reflectance isn't constant across the surface
- Function of both incoming direction and outgoing direction



Bidirectional Reflectance Distribution Function

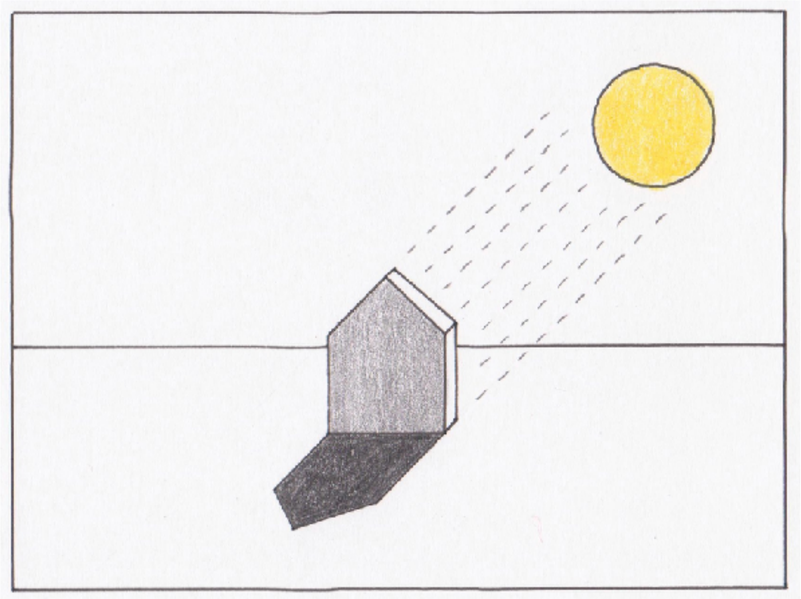
What About Shadows?

- Point lights cast hard shadows
- Area lights cast softer shadows
  - Umbra = area in full shadow
  - Penumbra = area in partial shadow



Simple Shadows

- For point lights, shadows are pretty simple
- *Do a visibility test from the point of view of the light!*
- Z-buffering can also be used for distance-based falloff



Coming up...

- More on lighting and shading

Lighting and Shading (cont'd)

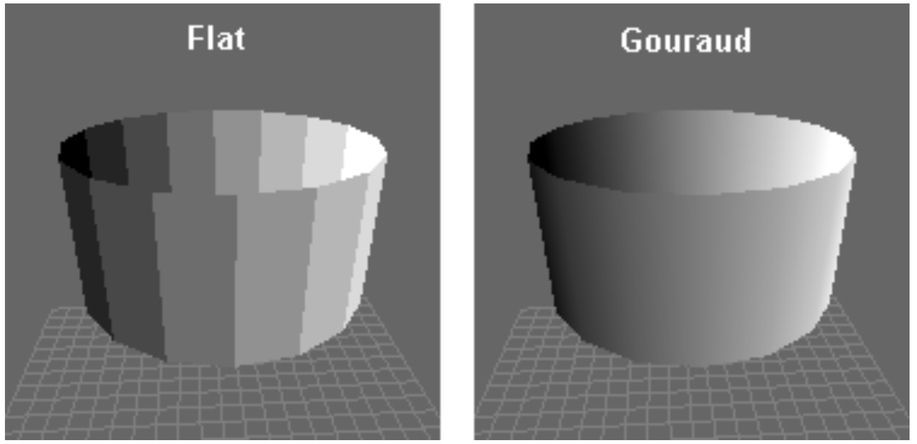
CS 355: Introduction to Graphics and Image Processing

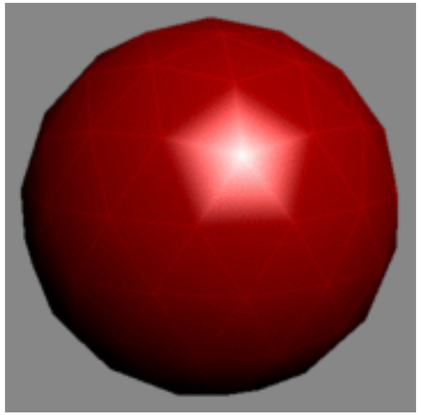
Flat Shading

- The entire polygon has the same normal, so it's all colored the same
- Leads to "flat shading"

Gouraud Shading

- Simple idea:
  - Compute shading at edges/ vertices and interpolate
  - Works well for diffuse
  - Doesn’t work as well for specular

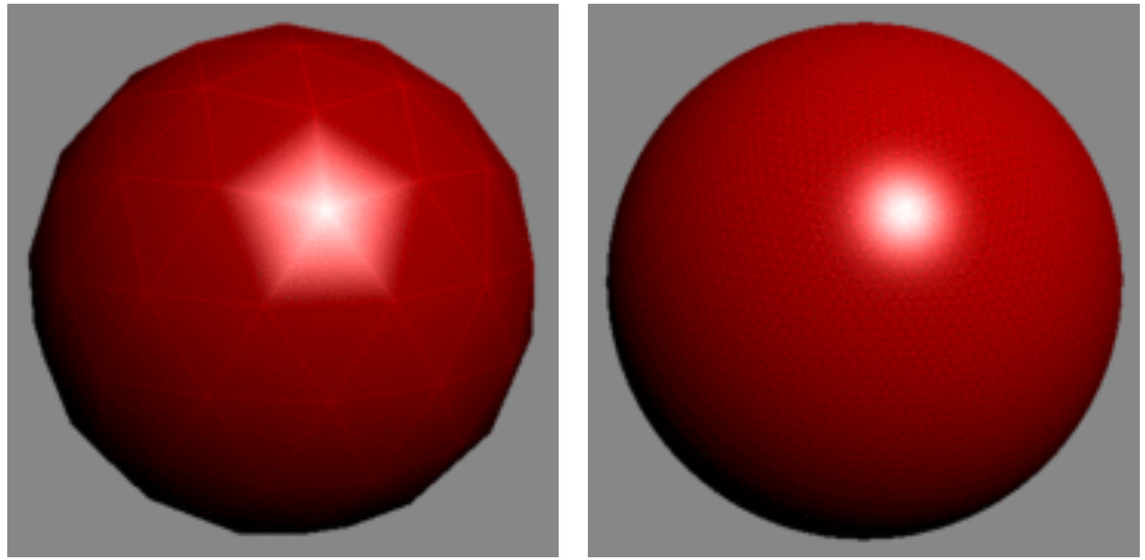




Henri Gouraud, 1971

Phong Shading

- Key idea:
  - Instead of interpolating the shading, *interpolate the normals*
  - Compute shading on a per-pixel basis using interpolated normal

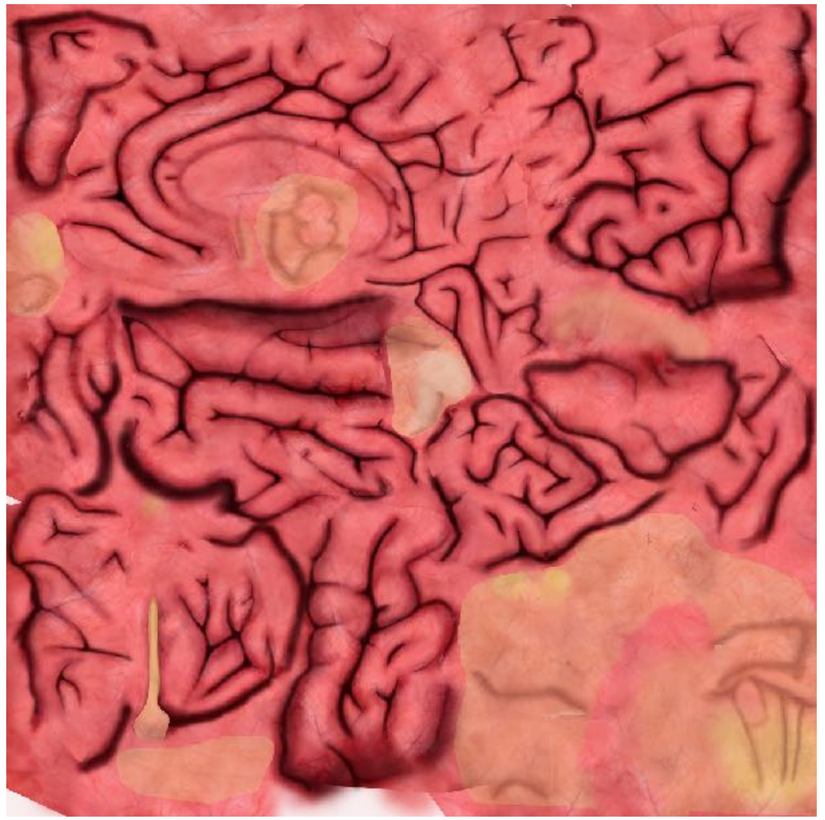


Bui Tan Phong, 1973

Texture Maps

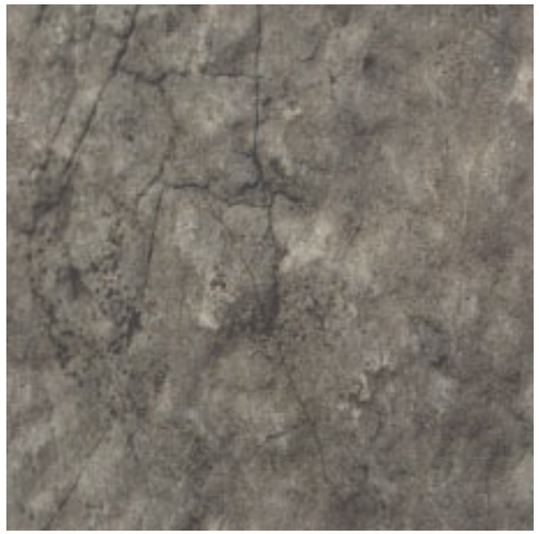
Texture Maps

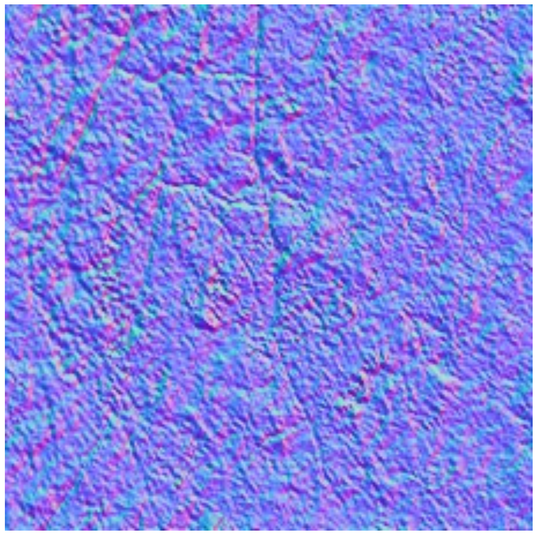
- Reflectance across a surface varies with position
- Map each vertex (x,y,z) to a point (u,v) in an image
- Interpolate across the polygon to interpolate the corresponding (u,v) positions in the image



Bump Maps

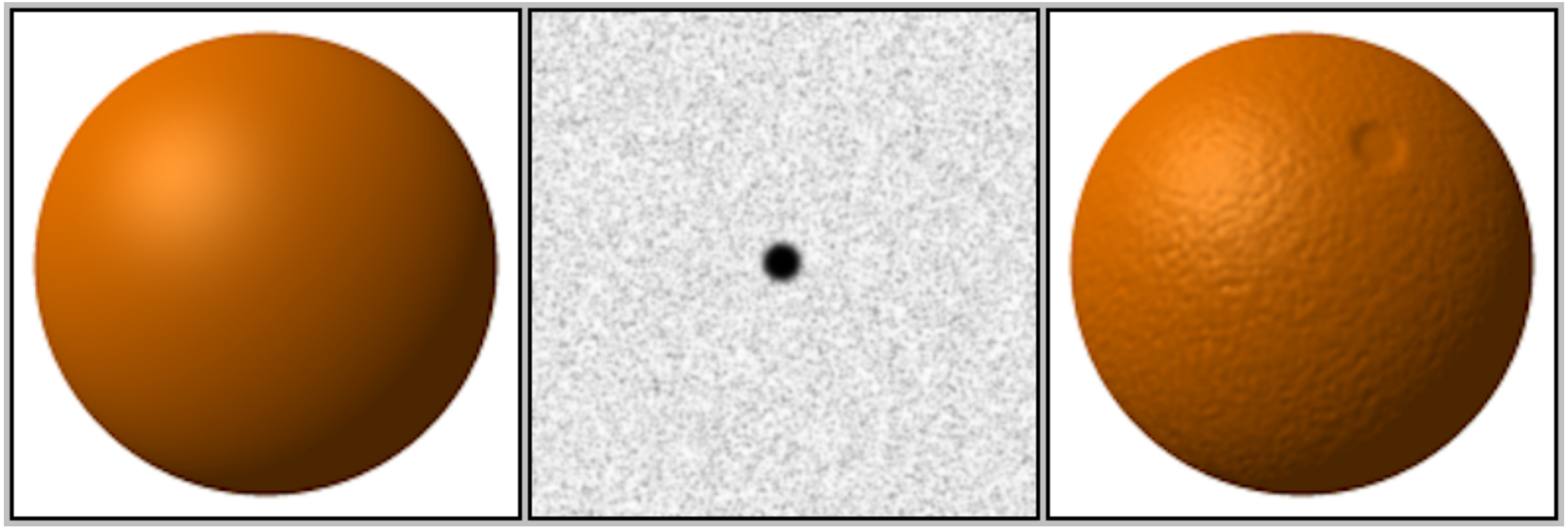
- Texture maps are only reflectance
- They don’t respond to changes in the lighting (“painted on”)
- Idea: use a *normal map*



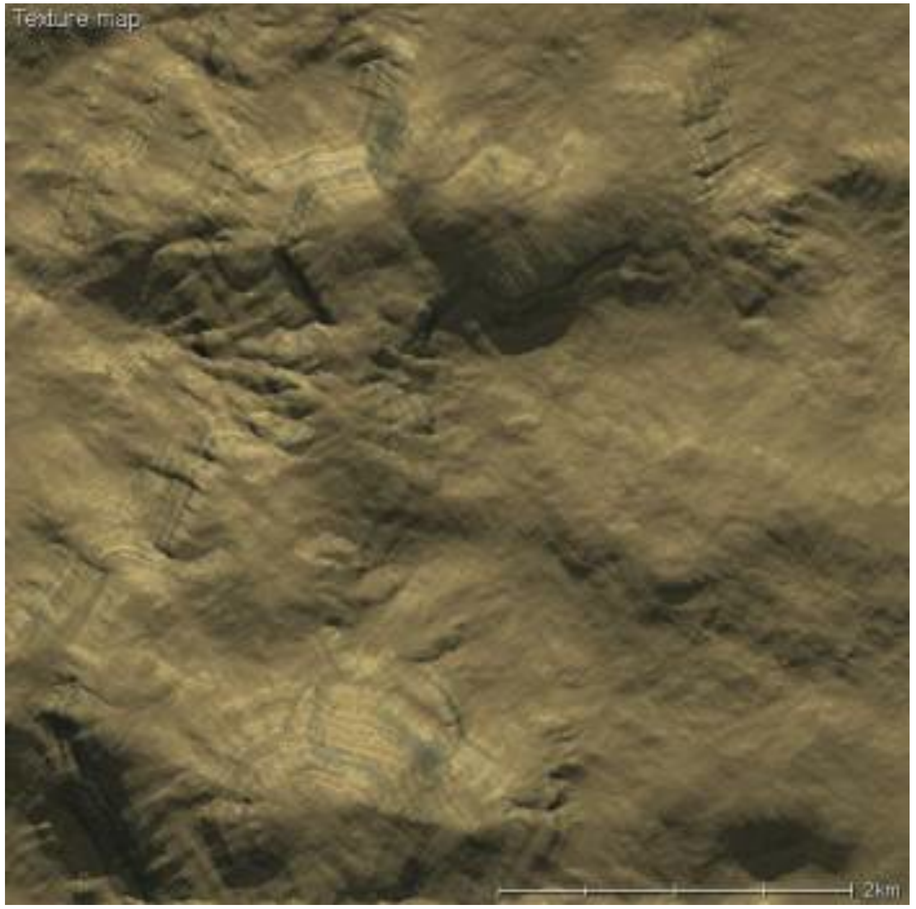


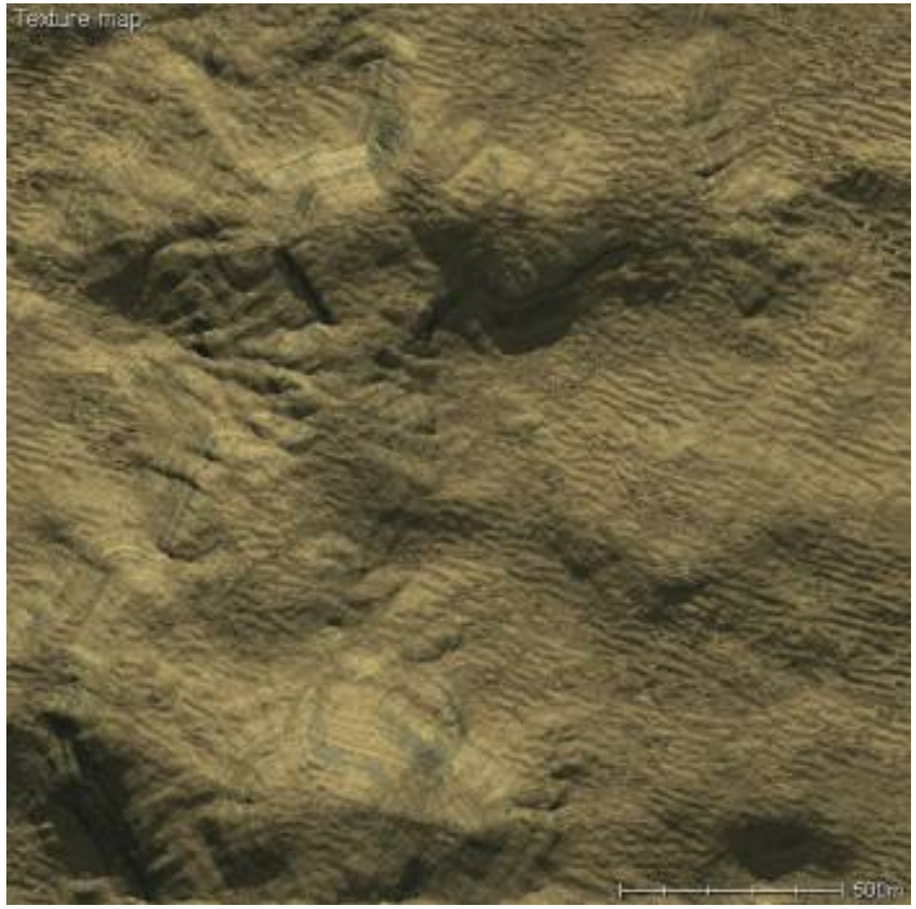
Jim Blinn, 1978

Bump Maps

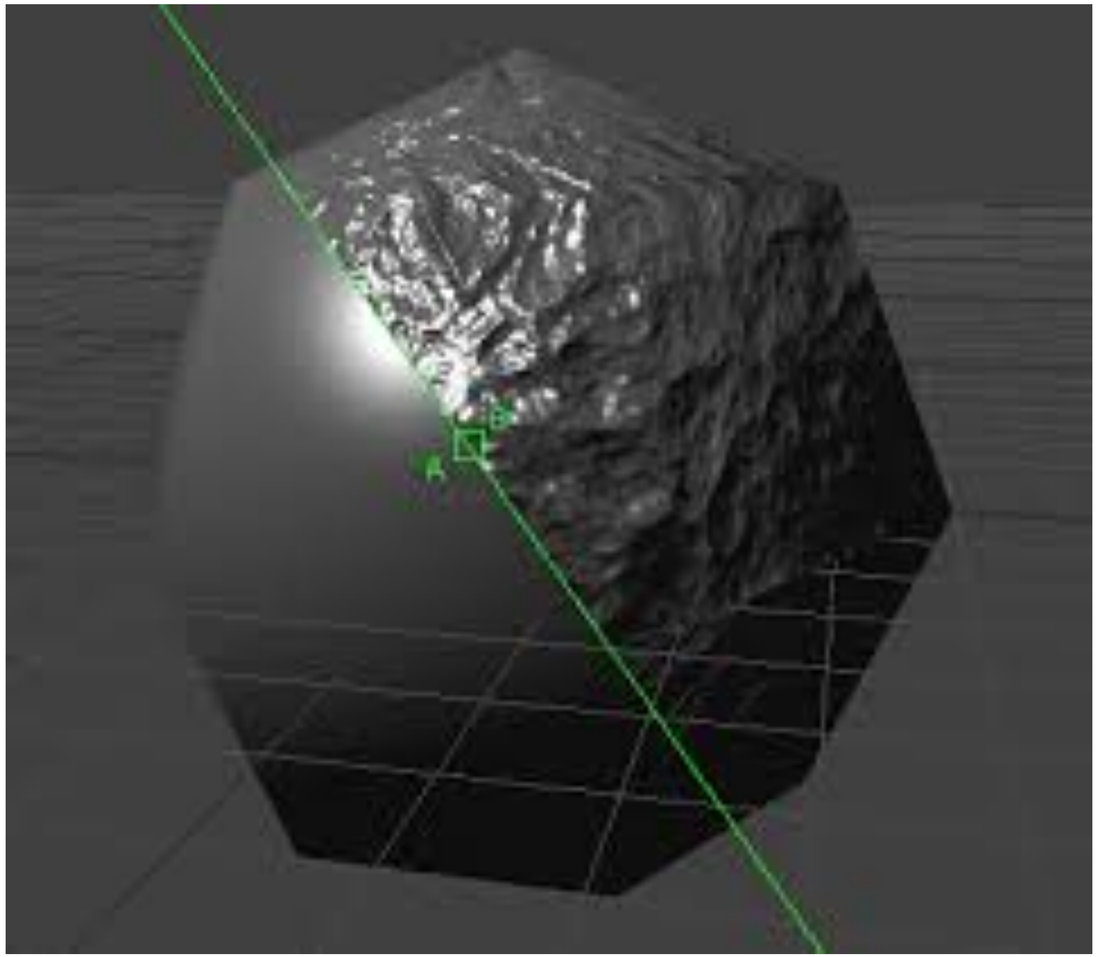


Bump Mapping

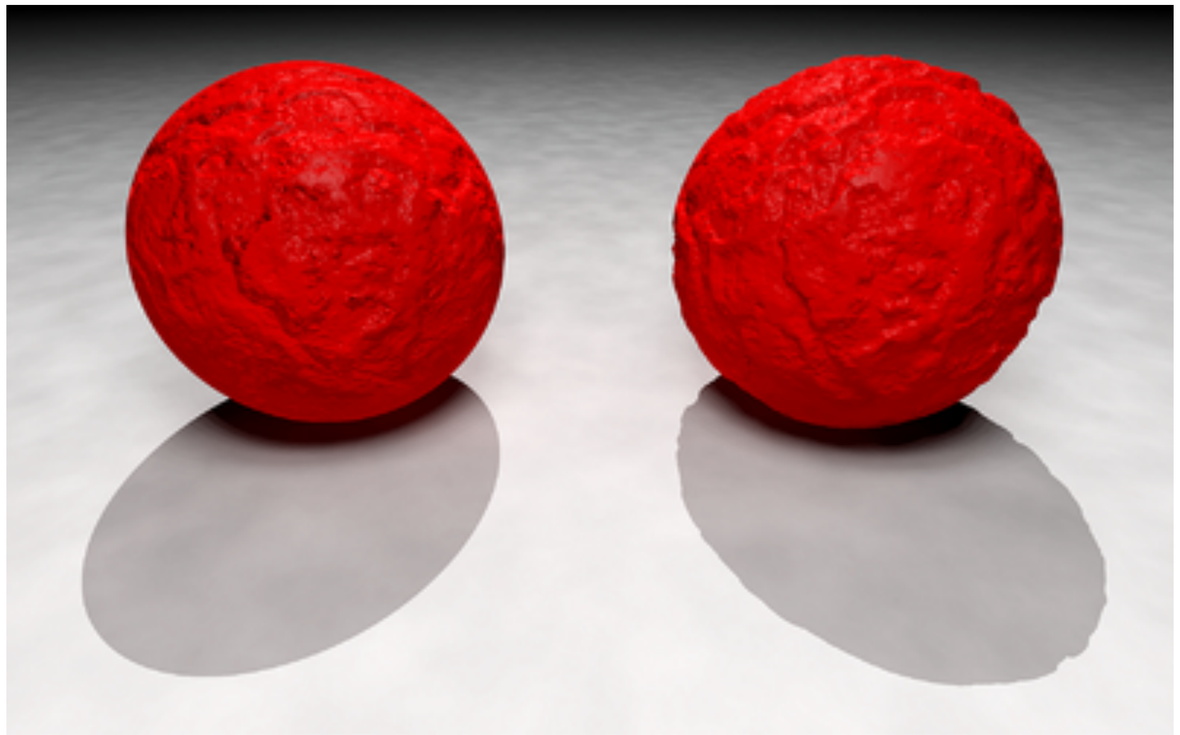




Bump Mapping



Bump Mapping

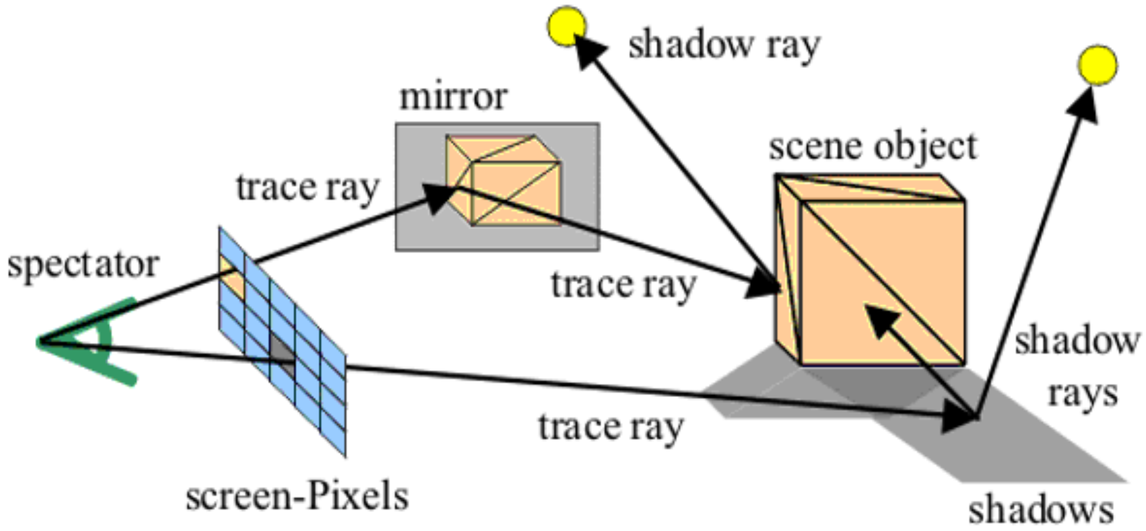


Ray Tracing

- What about things like
  - Reflections?
  - Refraction through transparent materials?

Ray Tracing

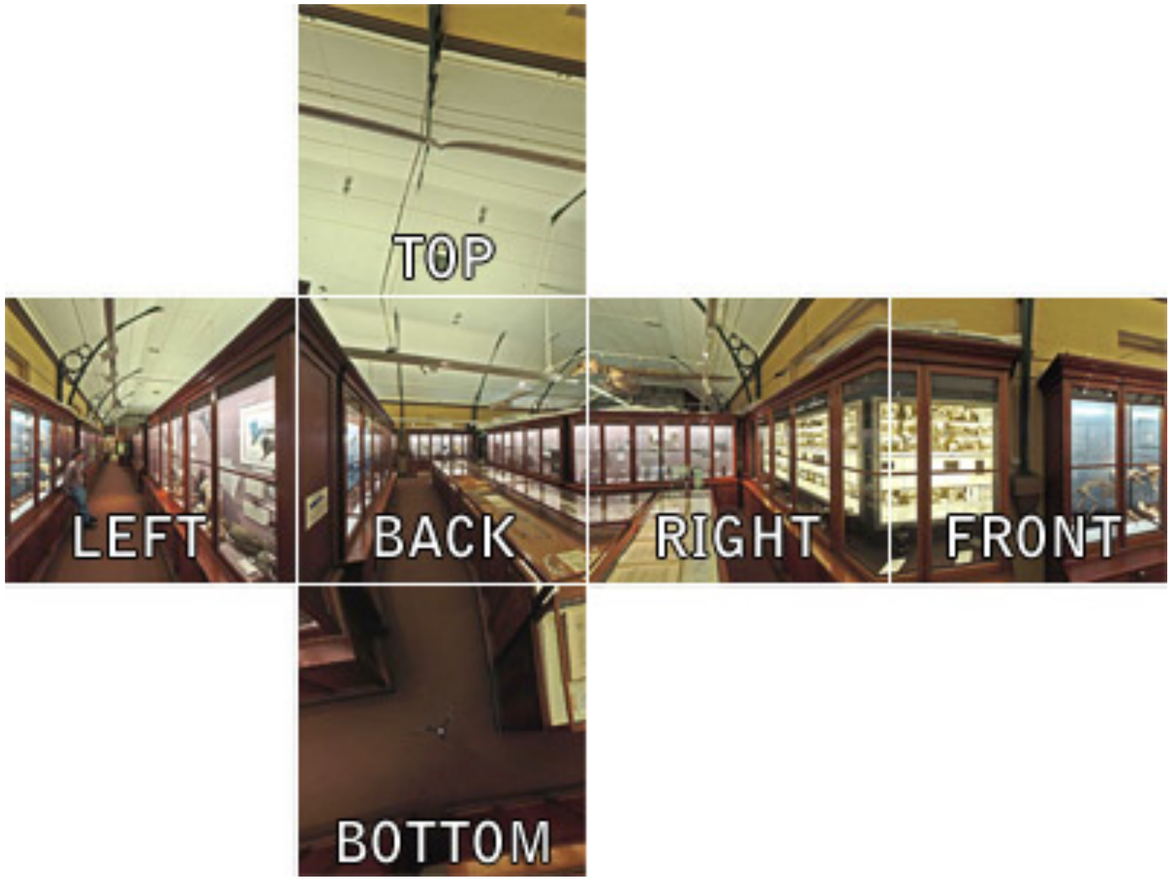
- Shoot a ray out through the pixel on the imaging plane
- When it hits something, ***recursively*** shoot more rays:
  - Towards light source(s) — direct lighting
  - Reflection ray (if applicable)
  - Refraction ray (if applicable)



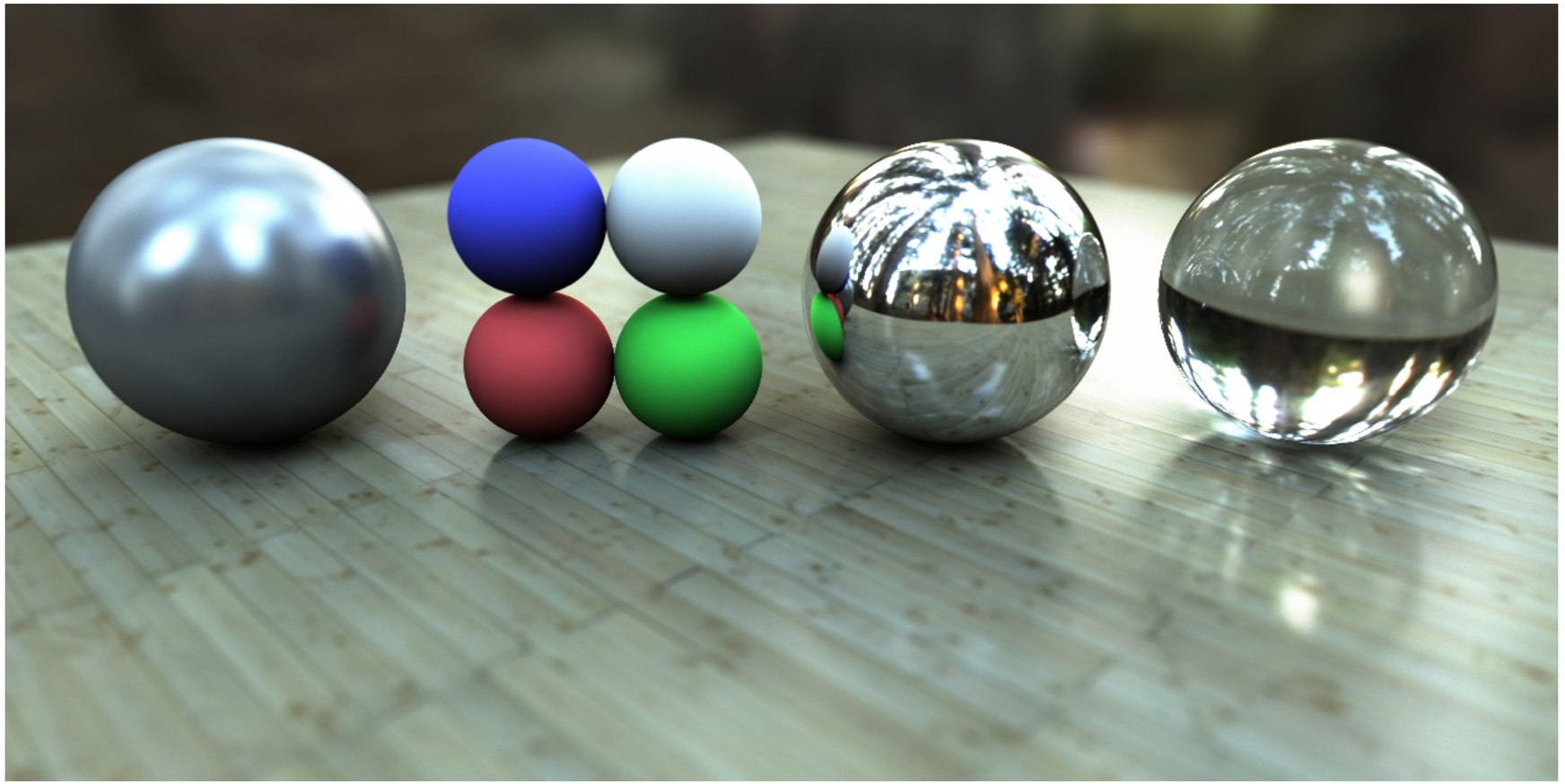
Turner Whitted, 1980

Environment Maps

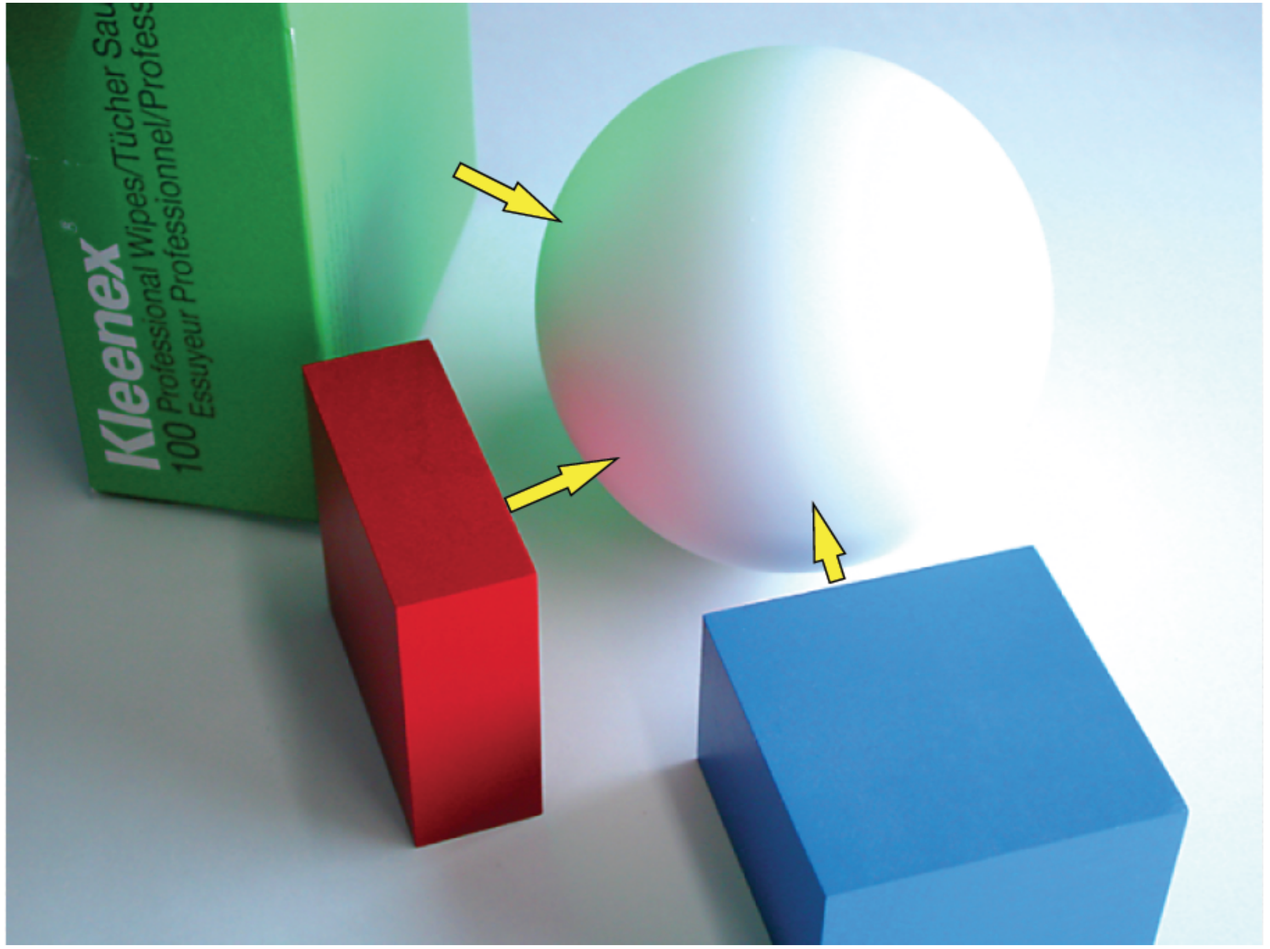
- What about rays that exit the scene?
- Can wrap the entire scene in an *environment map*
- Square, spherical, etc.



Environment Maps

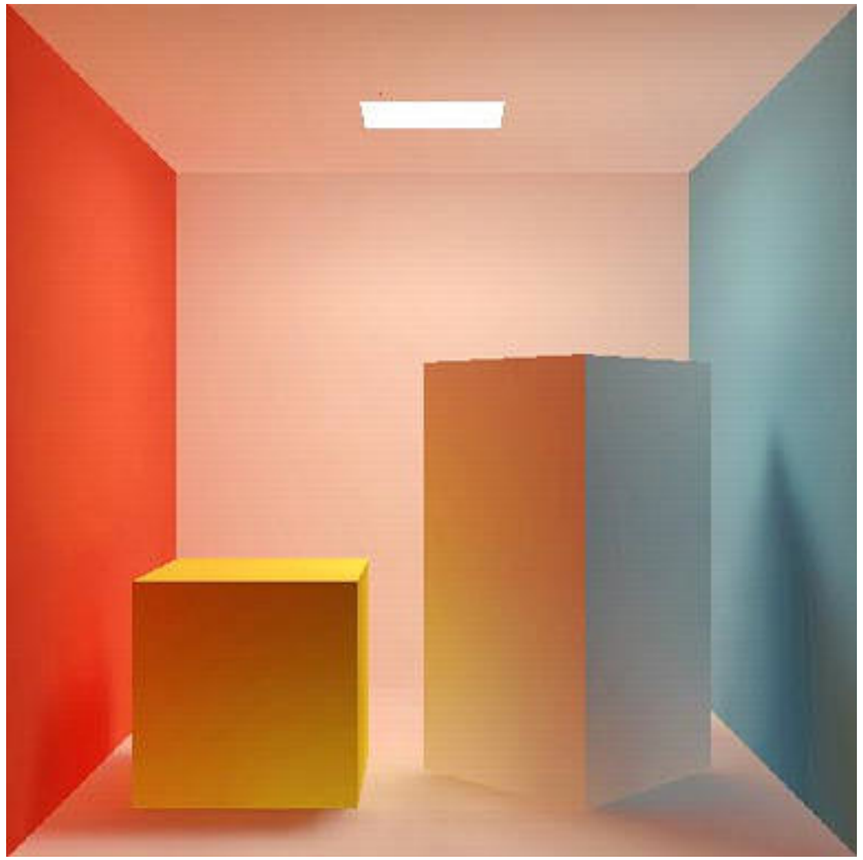


Diffuse Interreflections



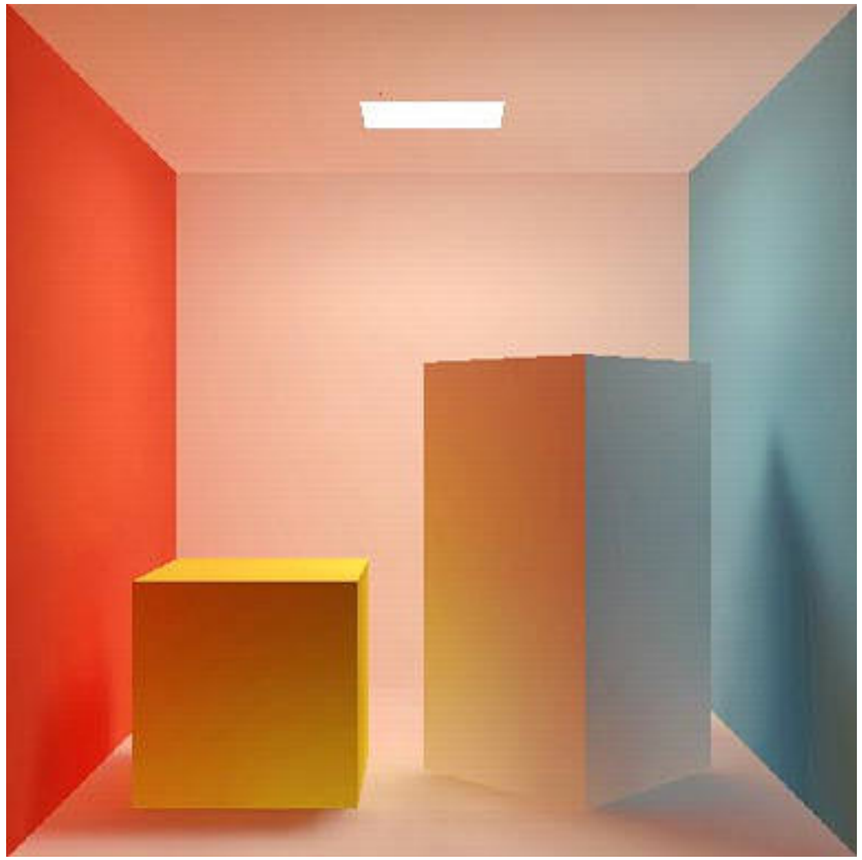
Radiosity

- Math involves integrals over *all angles* of light coming *into* and *off* of a surface
- Approximated using *importance sampling*
- Effectively traces lots and lots of rays — compute intensive



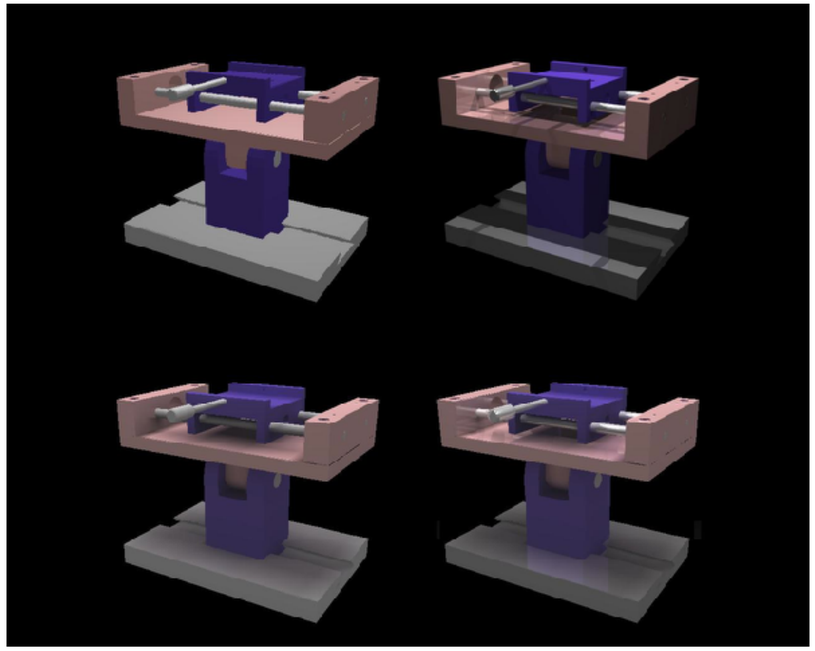
Radiosity

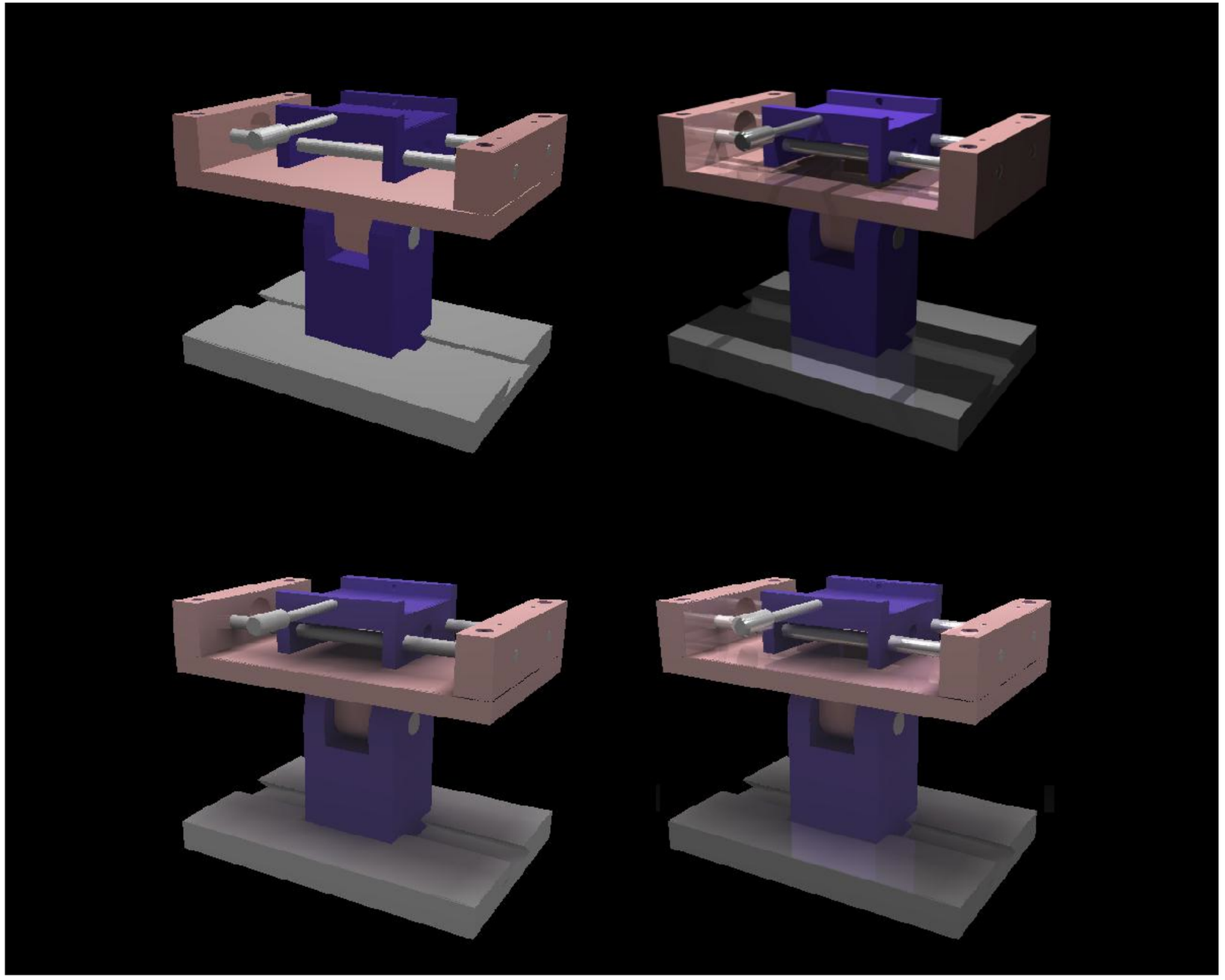
- Because radiosity involves diffuse light, it is independent of the viewpoint
- Can precompute ahead of time if objects and light sources *do not change*
- But if something other than the camera moves…



Combining Approaches

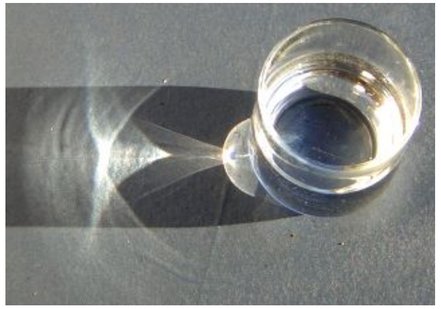
- Ray tracing is good for specular reflections, refraction, etc.
- Radiosity is good for diffuse interreflections
- Can combine methods to get both

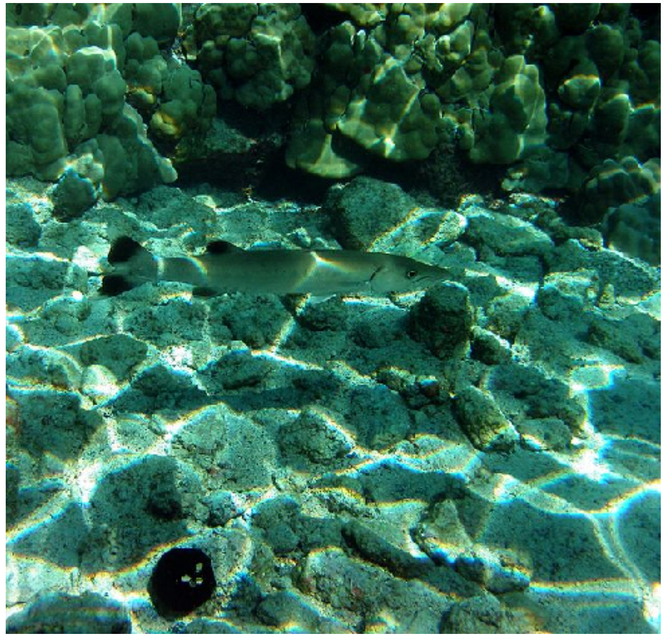




Forward Rendering

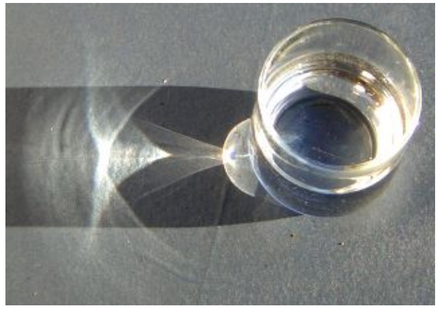
- Some phenomena can't be modeled well by going "backward" from the camera
  - ray tracing
  - radiosity
- Some require "forward" rendering of the light from the light source
- Example: "caustics"

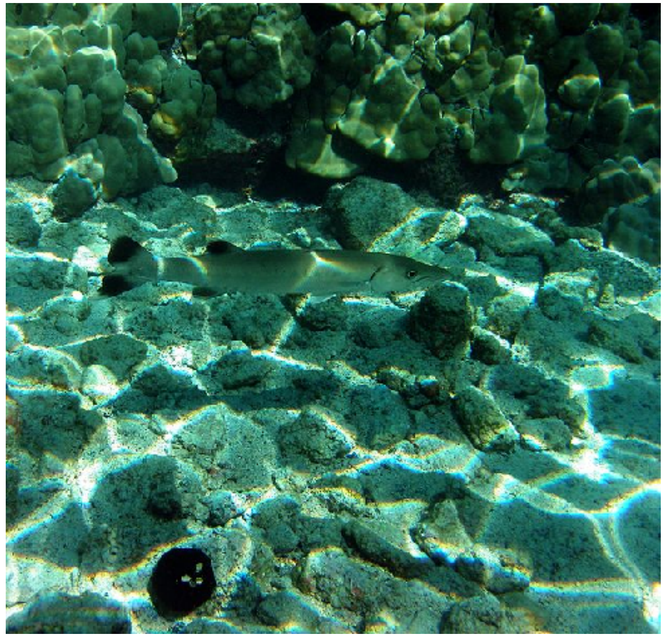




Forward Rendering

- Forward or hybrid methods:
  - Path tracing
  - Metropolis light transport
  - Photon mapping
- All try to *approximate* the "rendering equation" (Jim Kajiya, 1986)





More in CS 455…

- Ray tracing and variations
- Radiosity
- Other lighting models
- Curves and surface modeling
  (we'll talk a bit about this coming up)
- Animation (movement)
- Physical-based modeling
- And much more…

The Rest of the Semester…

- Physical cameras revisited
- Interpolation
- Image warping
- A bit more geometry…
- Curves and surfaces
- Frequency-domain processing

Cameras and Signal Acquisition

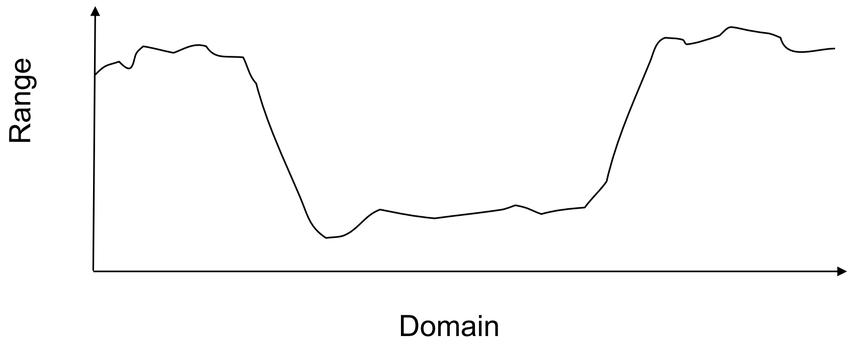
CS 355: Introduction to Graphics and Image Processing

Signals as Functions

- Digital signals can be thought of as *sampled functions*

- Domains:
  - Time (audio)
  - Space (images)
  - Both (video)

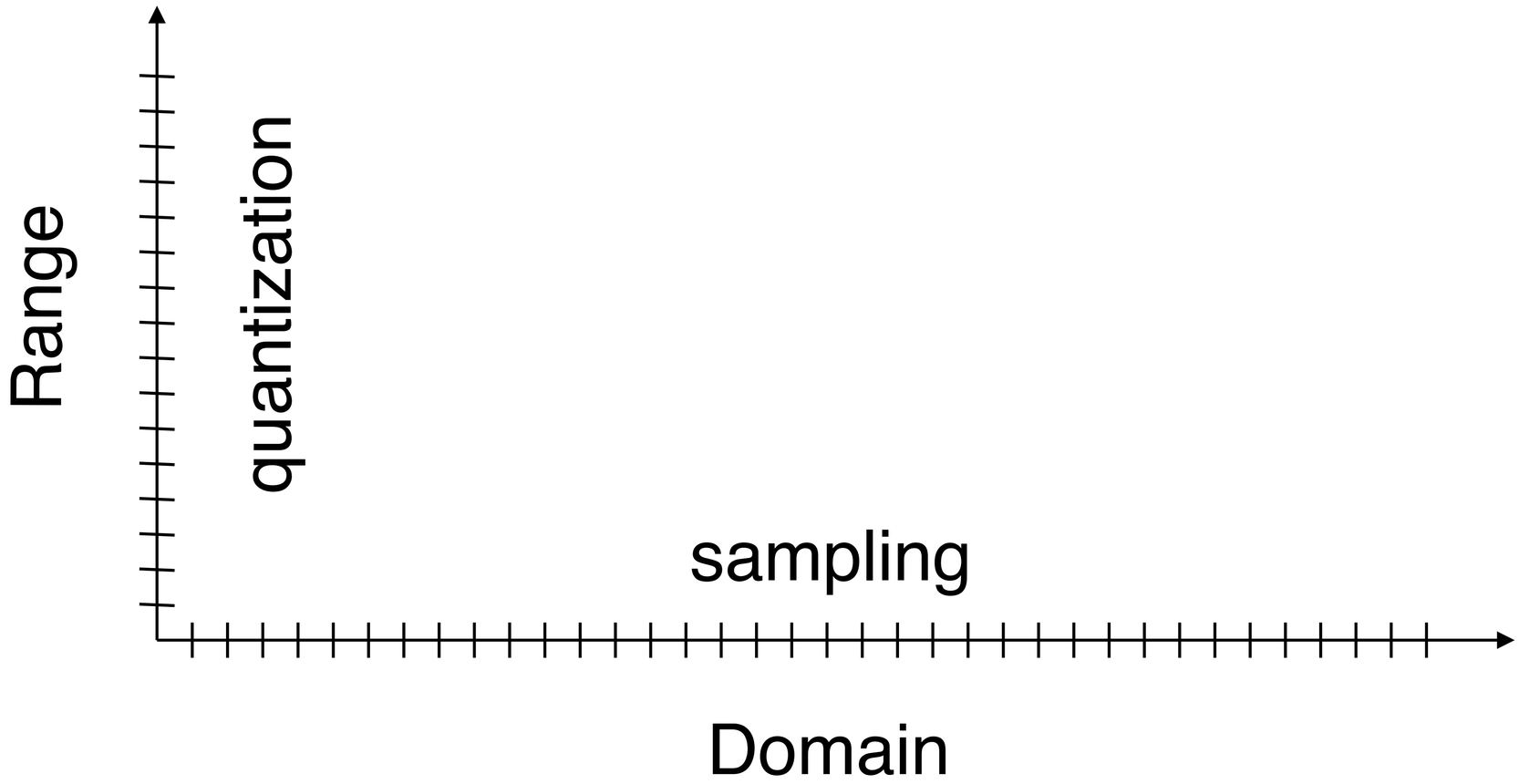
- Ranges:
  - Changing air pressure (audio)
  - Visible light (photographs, video)
  - Other properties
    (X-rays, MRI, range images, etc.)



$$f(t)$$

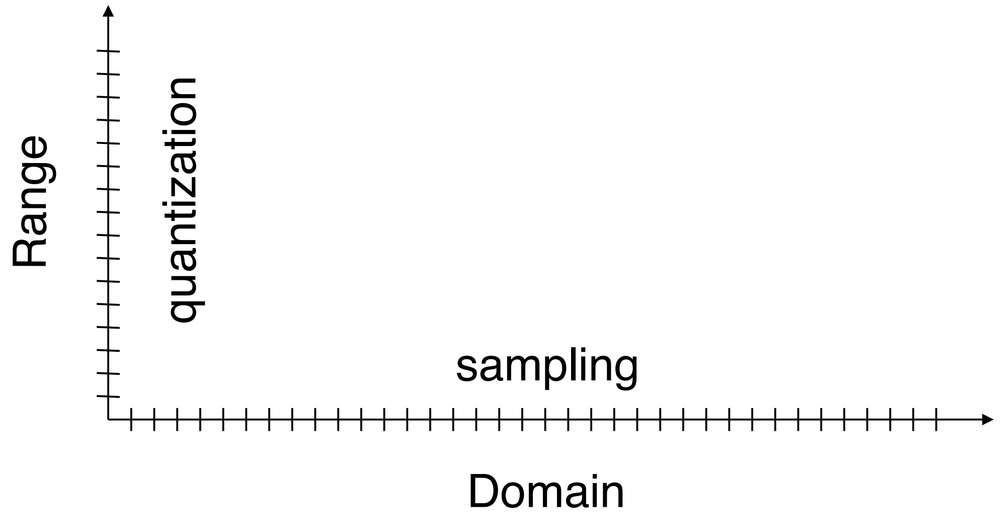
$$I(x, y)$$

Sampling vs. Quantization



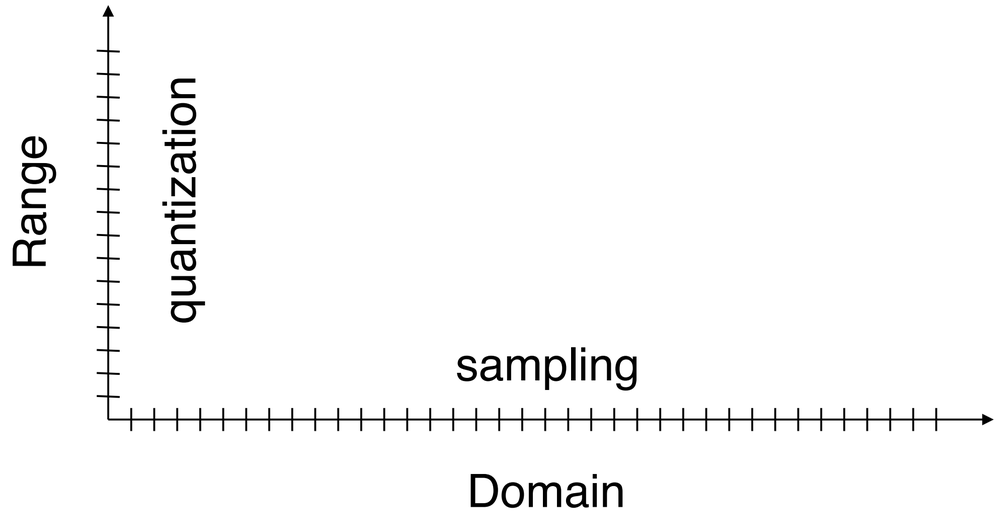
Sampling

- Samples per unit length, area, etc.
- Often expressed as
  - rate
  - spacing
  - density



Quantization

- *Levels of precision* in each sample
- Usually
  - number of levels
  - number of bits



Sampling vs. Quantization

- 600 dots per inch
- black and white images
- 256-level grey
- 8-bit grey

- 30 frames per second
- 24-bit color
- 44.1 KHz audio
- 16-bit audio

Storage

- We usually store digital signals (including images) as arrays
  - Audio: 1-D domain, 1-D array of values (PCM)

90	50	8	…	42

  - Images: 2-D domain, 2-D array

88	86	8	…	9
91	92	10	…	7
87	91	9	…	8
⋮	⋮	⋮	⋱	⋮
90	89	11	…	8

Storage

- In memory: usually just arrays
  - Again, be careful of (x,y) vs. row-major ordering
- On disk: may be something else entirely
  - Tiled storage (think virtual memory)
  - Hierarchical/Interlaced
  - Compressed
- Headers: EXIF, compression settings, etc.

How do we get images?

Acquisition Devices

- Aperture 
- Scanning
- Sensor 
- Quantizer
- Output storage medium

Apertures

- Pixels aren't point samples
- Total light over an *area* of the visible scene
- Controlled by the camera's iris (photographers: F-stop)
- Also caused by physical sensor area (pixel's area on the camera's sensor)

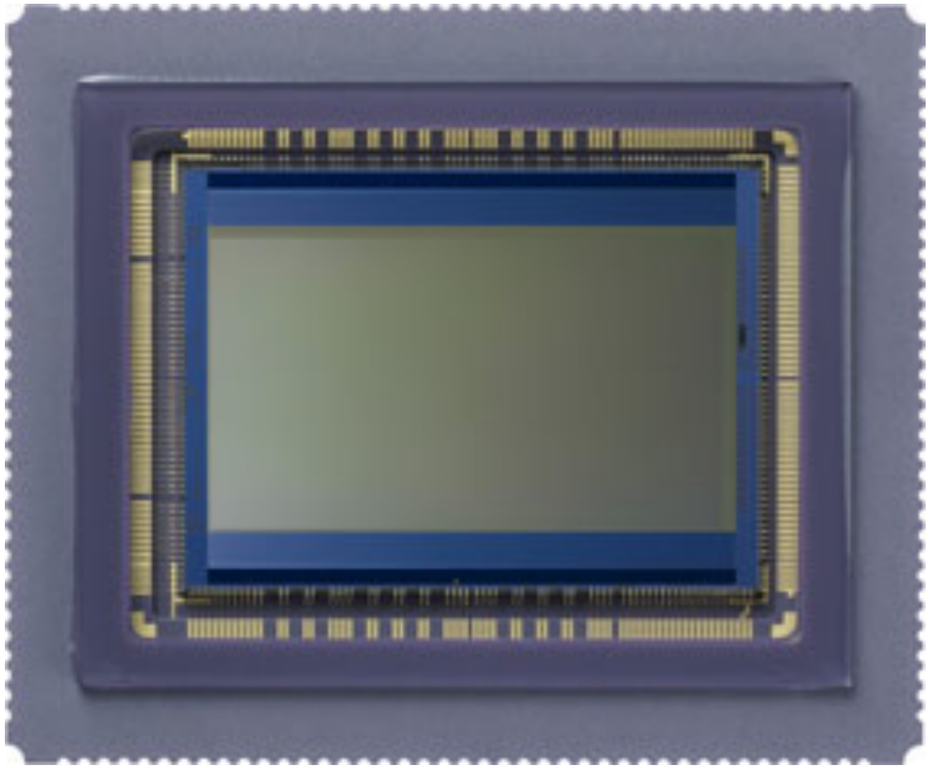
Apertures

- Sampling and size of aperture determine resolution
  - Smaller apertures = better resolution
  - Larger apertures = worse resolution
- Lenses allow a physically larger aperture to act as an effectively smaller one

Resolution

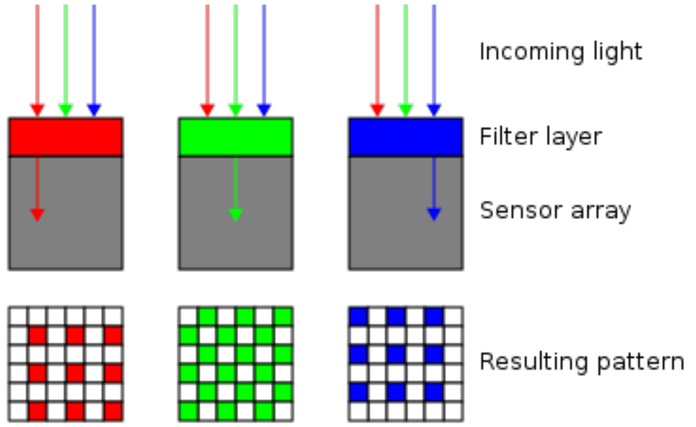
Sensors

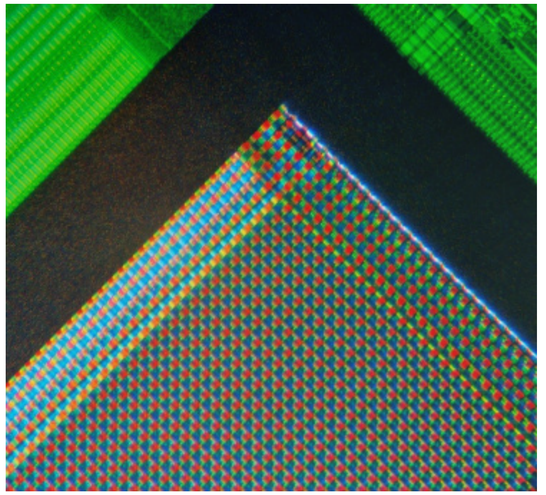
- Converts light (photons) to chemical and/or electrical response
- Examples
  - Silver halide crystals (film)
  - Photoreceptors in our eyes (rods, cones)
  - Charge-coupled device (CCD)
  - CMOS arrays



Bayer Patterns

- Most commercial-grade cameras sample only one color per pixel
- Small colored filter over each sensor element
- 16 megapixels =
  - 8 megapixels green
  - 4 megapixels red
  - 4 megapixels blue
- You get interpolated combination





Noise

- Unavoidable random fluctuations from “correct” value
- Can usually be modeled as a statistical distribution with mean at the “correct” value
- A measured sample will vary from that mean according to the distribution std. dev.

μ

σ

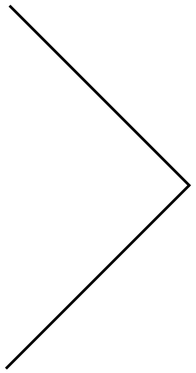
Signal-To-Noise Ratio

- Measure of how “noise free” a signal is

$$\mathrm{SNR} = \frac{\mu}{\sigma}$$

Sources of Noise

- Quantum nature of light
- Sensor inhomogeneity
- Electrical fluctuations
- “Background” noise



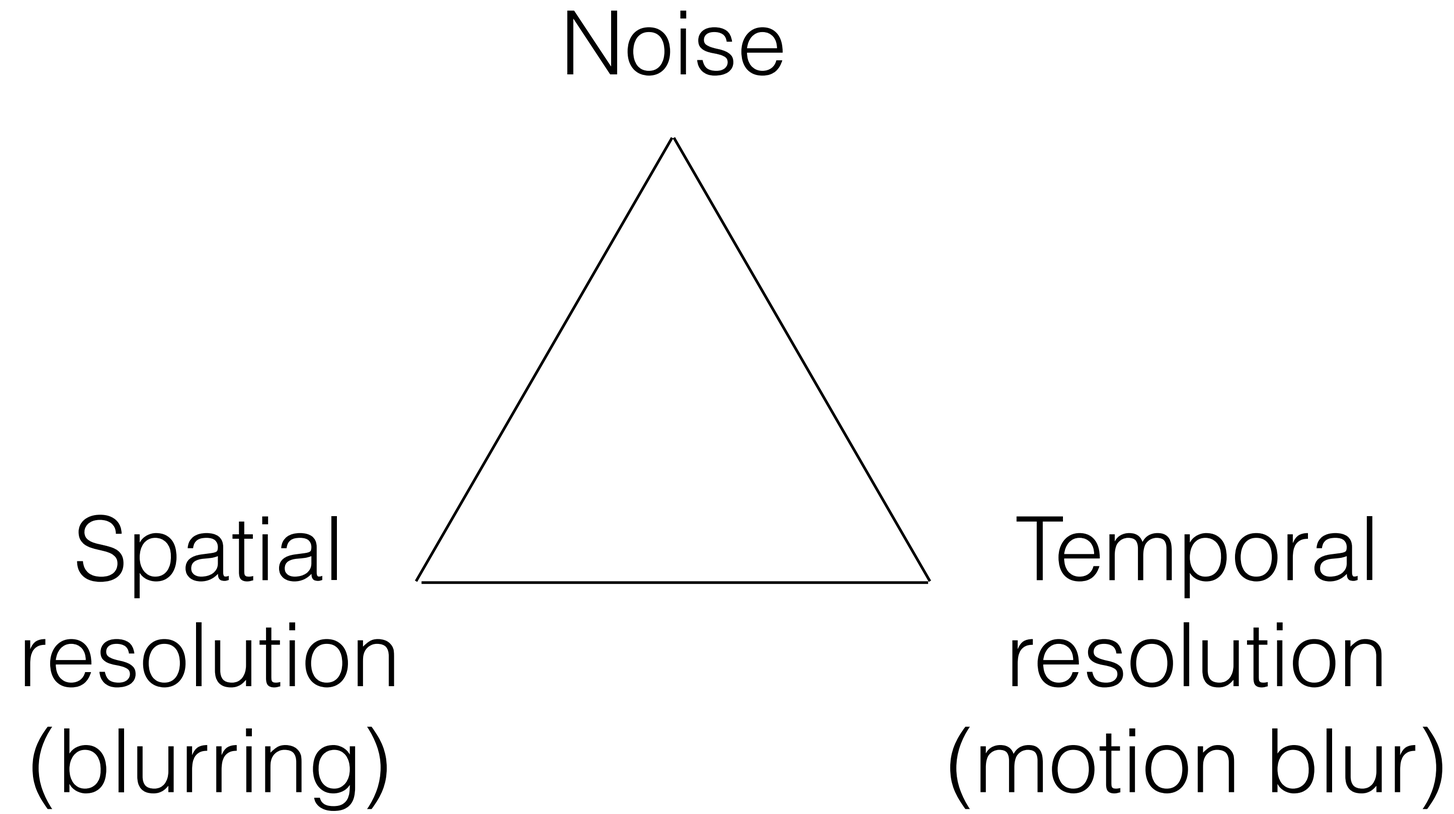
May not be random

Shot Noise

Reducing Shot Noise

- The only way to reduce quantum noise is to *collect more light*
  - Turn up the source
  - Larger aperture
  - Collect for longer
- What are the tradeoffs?

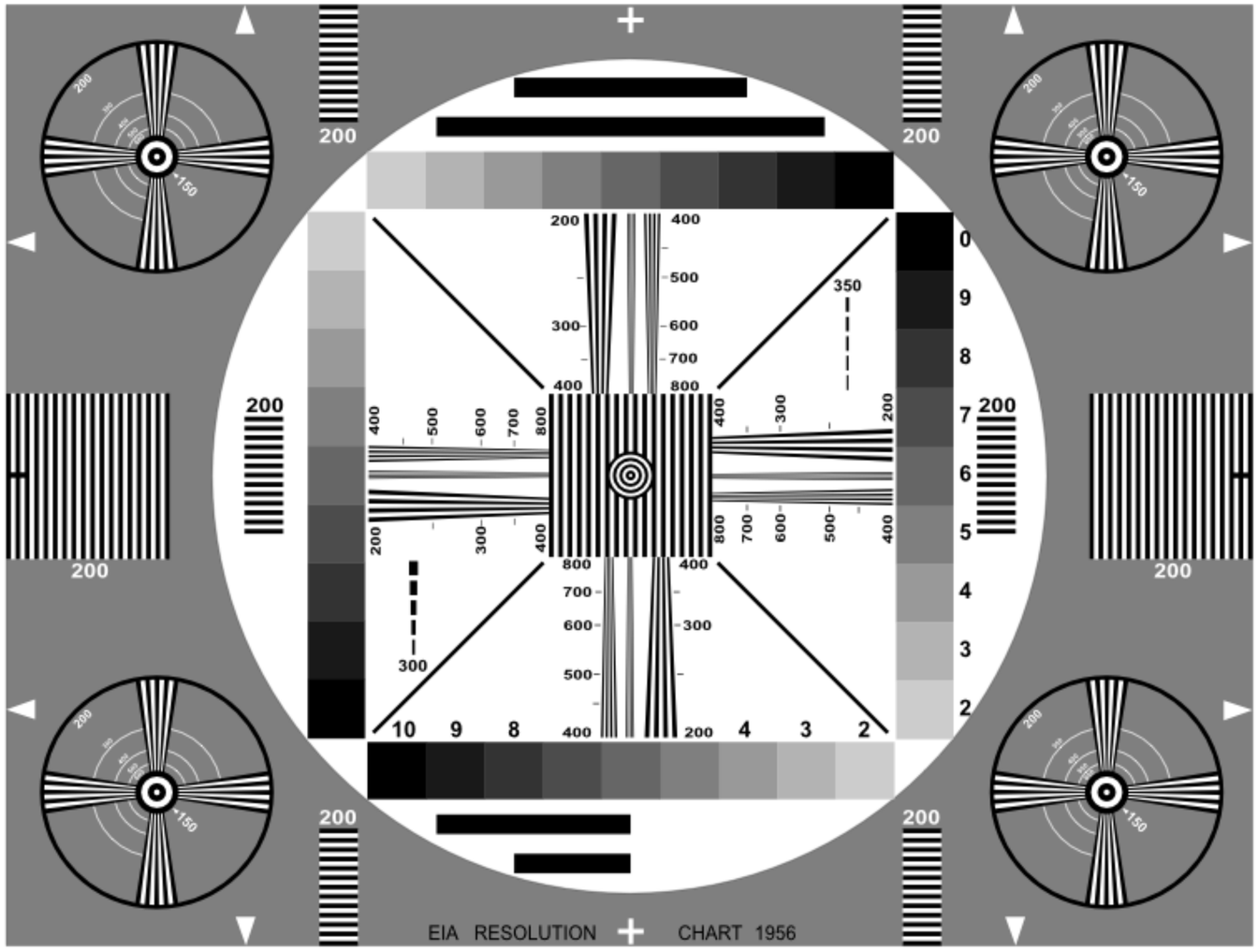
Fundamental Tradeoff



Measuring Resolution

- One way is to use alternating black/white lines with fixed spacing
  - Increase the density until you can’t see the separate lines
    - Gradually blurs to grey
    - Stop when half the original contrast
  - Units: line pairs per millimeter

Measuring Resolution



Sampling Revisited

- How much sampling is enough?
  - *Shannon Sampling Theorem*:
    twice the highest frequency in the signal (in theory)
  - *Nyquist rate*
- What happens if you sample above this?
  - Avoids dangers of theoretical limits
  - Better for intermediate processing
- What happens if you don't sample enough?
  - Aliasing (false low-frequencies components appear)
  - In images this causes *Moiré patterns*

Different kind of aliasing
than jaggies, but related

- *Insufficient sampling during acquisition introduces flaws that cannot be corrected in later processing*

Moiré Patterns

Camera Problems

- Noise
- Spatial blur
- Motion blur
- Bayer sampling artifacts
- Lens distortion
- Chromatic aberration
- Brightness
- Contrast
- Color balance
- Tone mapping
(color responses)

Coming up…

- Interpolation of discrete samples

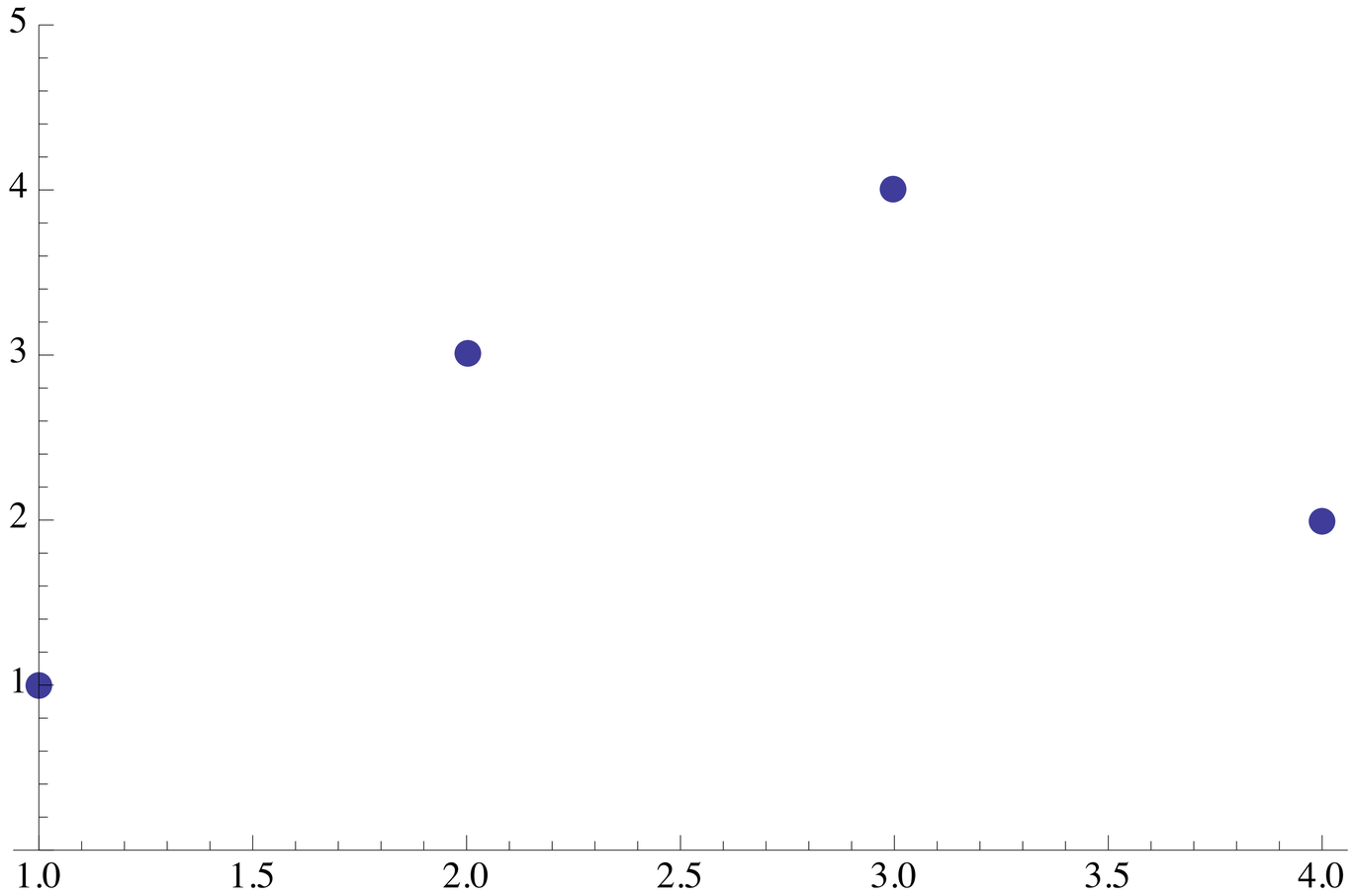
Interpolation

CS 355: Introduction to Graphics and Image Processing

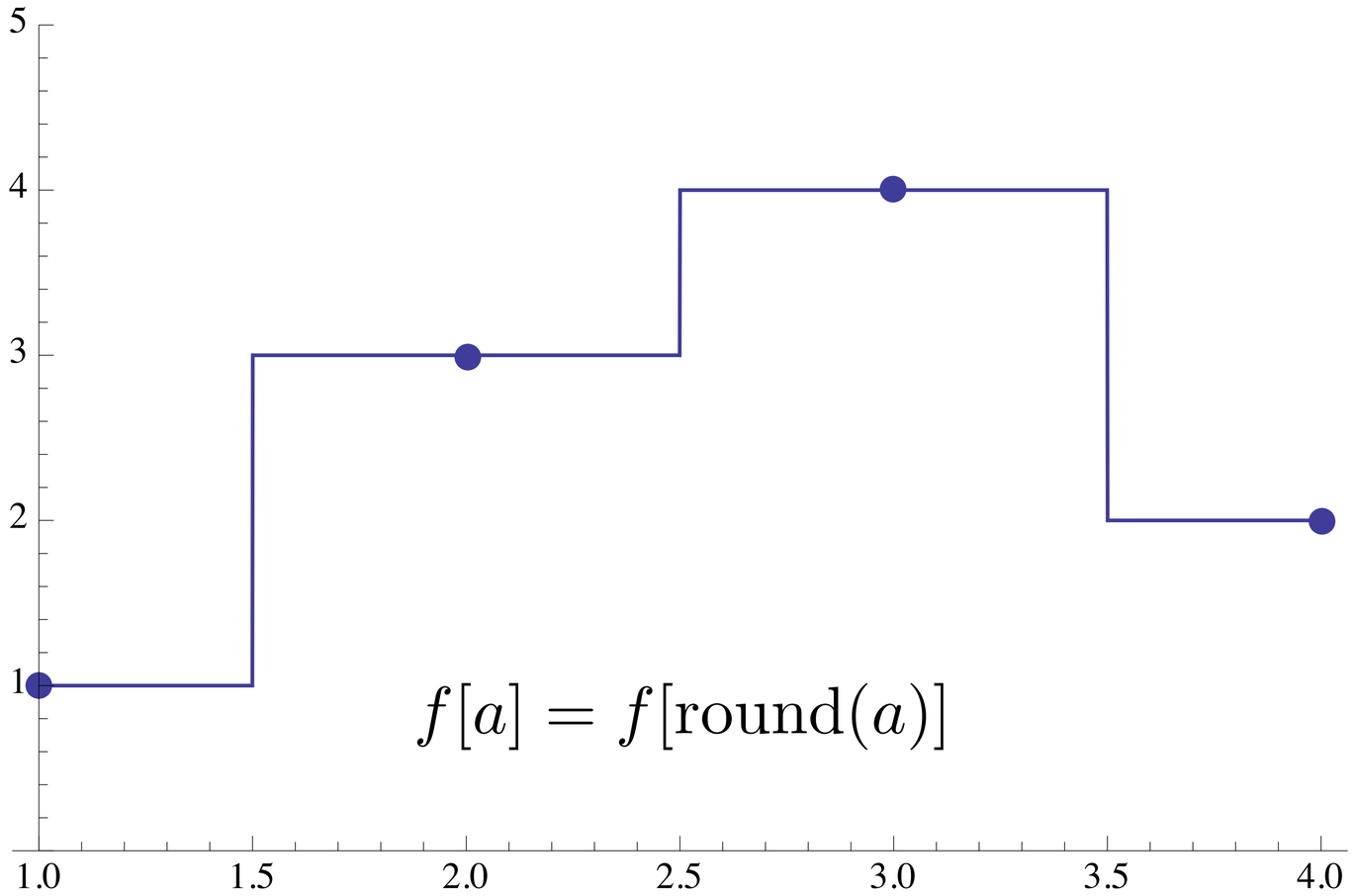
Interpolation

- Used to estimate a function between known sampled values
- Digital signals to analog
  - Audio playback
  - Image display
- Spatial image manipulation
  - Changing size
  - Rotation
  - Warping
- Generate smooth geometric models between defined points
  (we'll come back to this later)

Interpolation

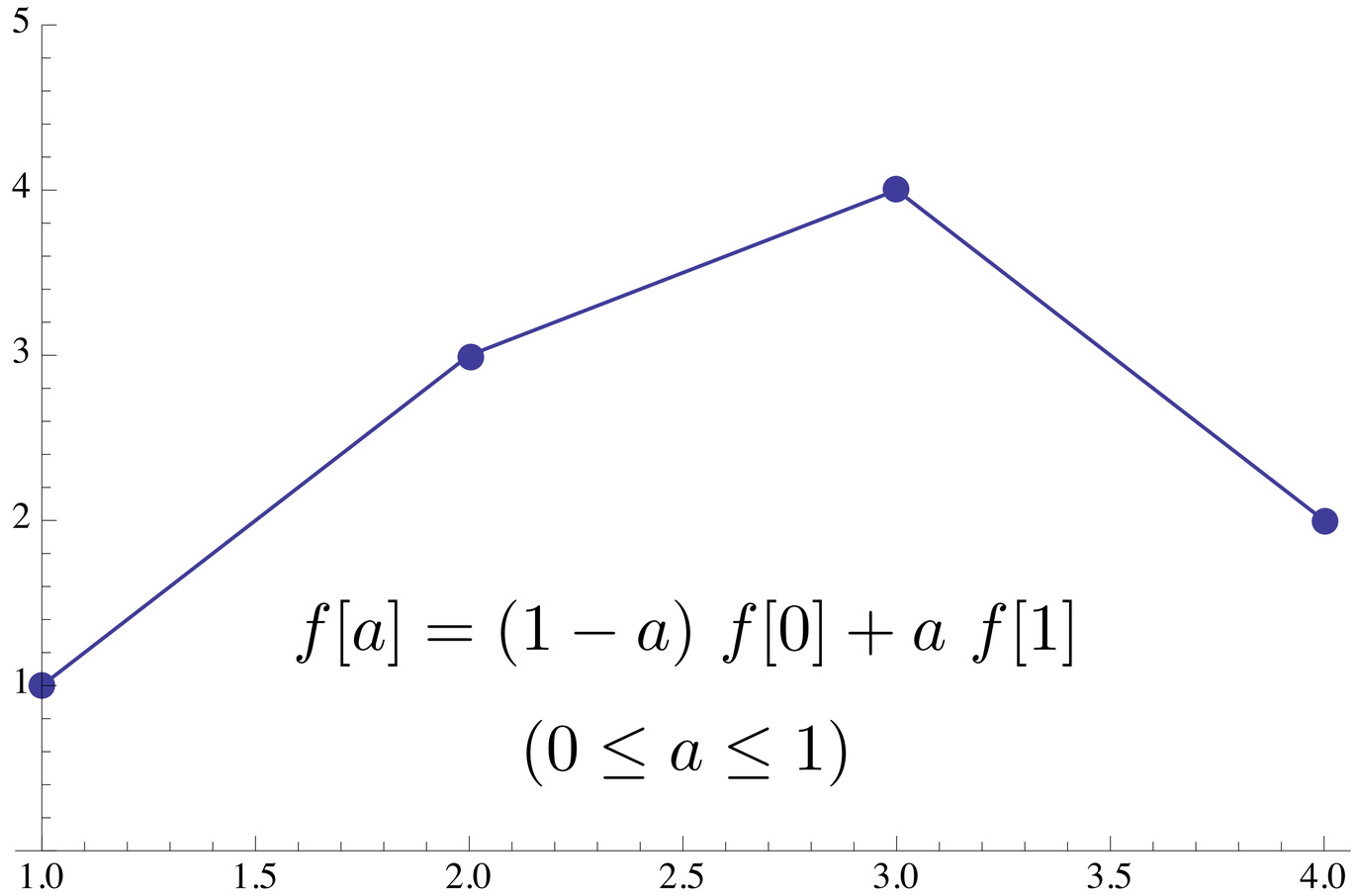


Nearest Neighbor



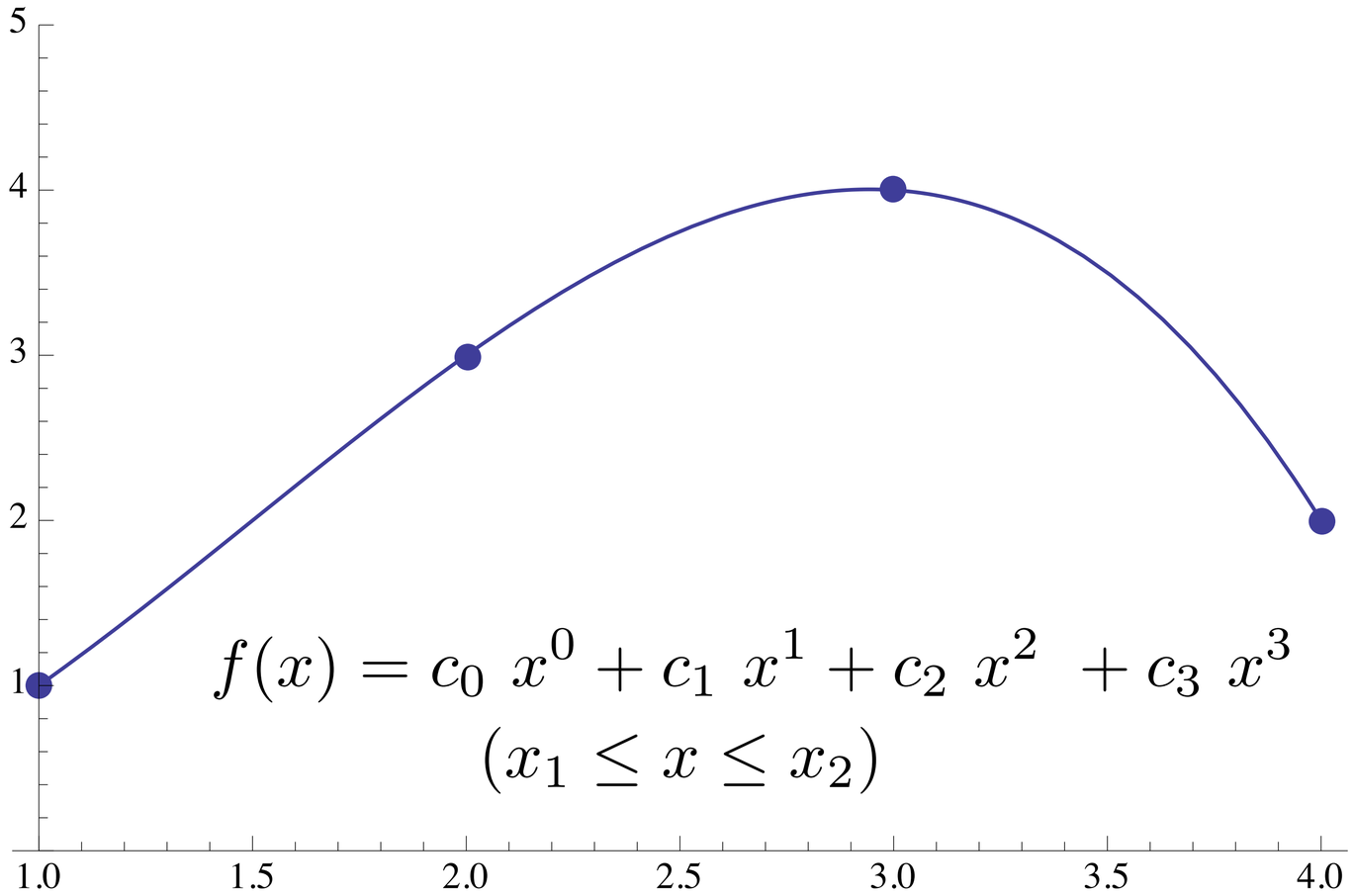
0th order

Linear Interpolation



1st order

Cubic Interpolation



3rd order

Cubic Interpolation

$$f(x) = ax^3 + bx^2 + cx + d$$

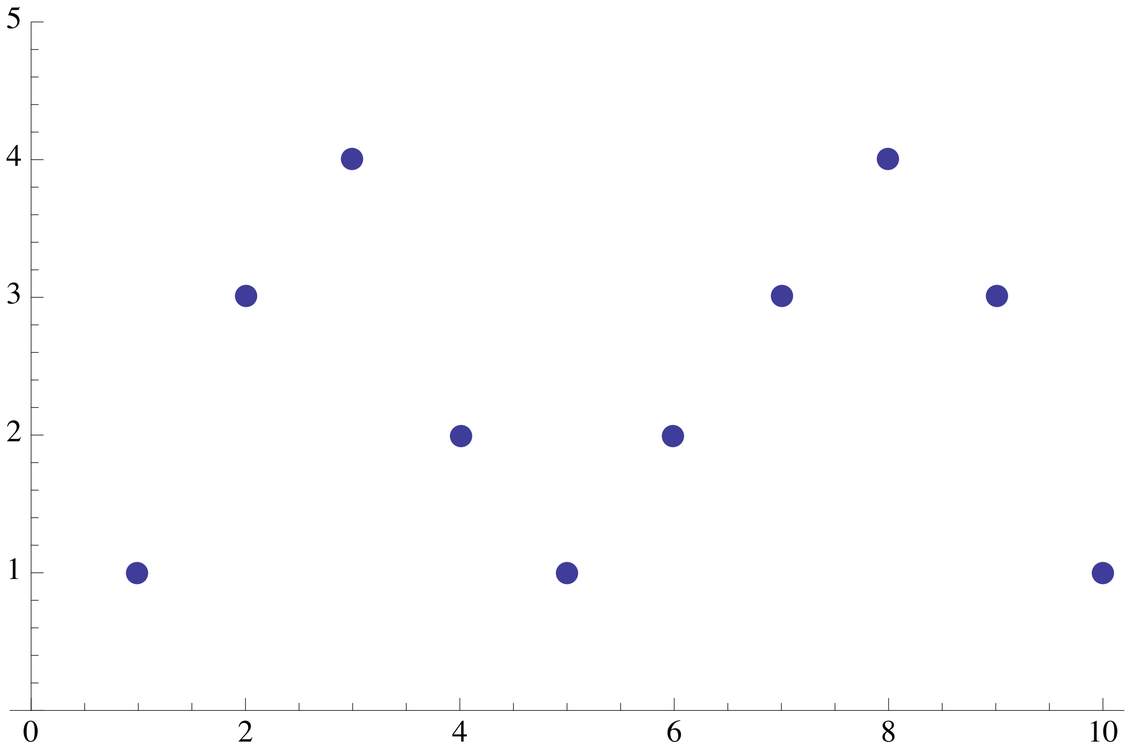
$$(x_1 \leq x \leq x_2)$$

$$\begin{aligned}
f(x_0) &= a\ x_0^3 + b\ x_0^2 + c\ x_0 + d \\
f(x_1) &= a\ x_1^3 + b\ x_1^2 + c\ x_1 + d \\
f(x_2) &= a\ x_2^3 + b\ x_2^2 + c\ x_2 + d \\
f(x_3) &= a\ x_3^3 + b\ x_3^2 + c\ x_3 + d
\end{aligned}$$

$$\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & x_0^3 \\ x_1^0 & x_1^1 & x_1^2 & x_1^3 \\ x_2^0 & x_2^1 & x_2^2 & x_2^3 \\ x_3^0 & x_3^1 & x_3^2 & x_3^3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \end{bmatrix}$$

Solve for a, b, c, d and plug into function

Longer Sequences

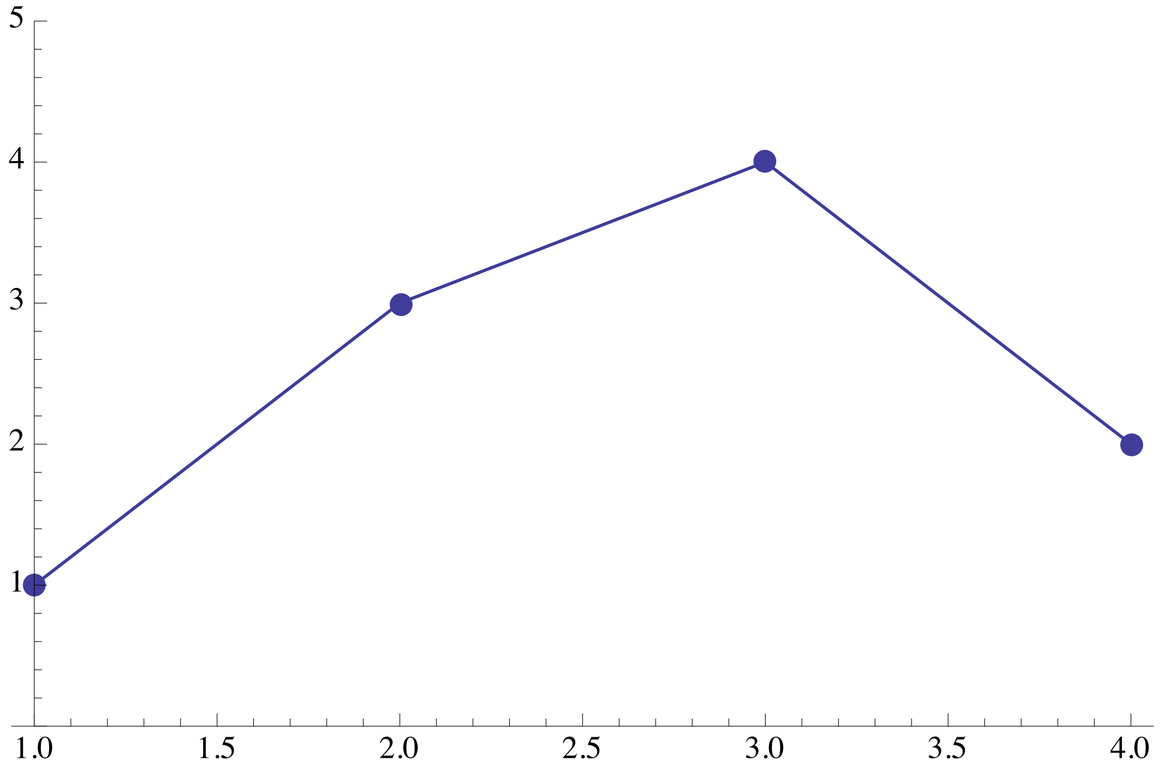


0th - nearest neighbor

1st - use one sample on both sides

3rd - use two samples on both sides

Continuity



0th order: function is continuous
1st order: slope is continuous
nth order: nth derivative is continuous

2D Images

- Extend ideas of single-value interpolation
  - Nearest neighbor
  - Linear
  - Cubic
  - …

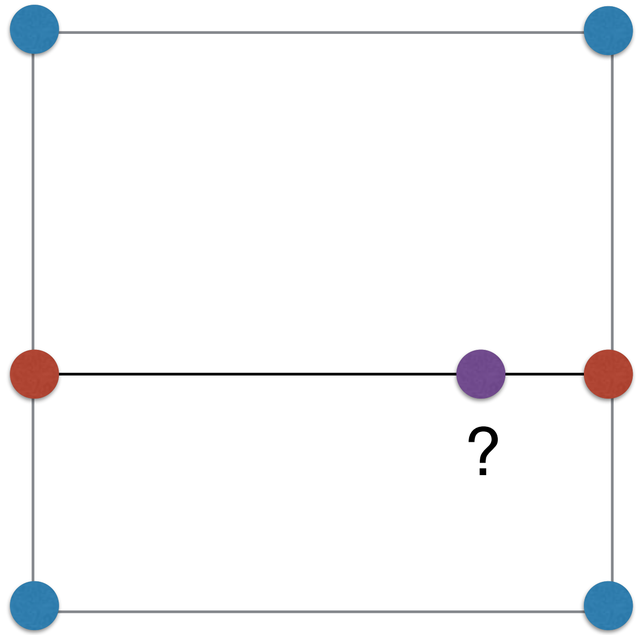
Nearest Neighbor

- Same idea as in 1-d:
  Round off to nearest pixel
- Also called “pixel replication”
- Big blocky pixels

Bilinear

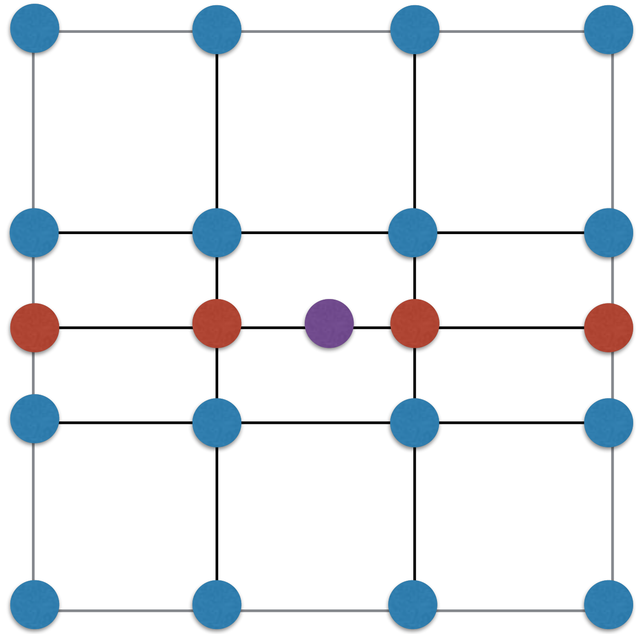
- Very common approach:
  - Interpolate vertically
  - Interpolate horizontally

  (or vice versa)
- For linear interpolation, this is called *bilinear*



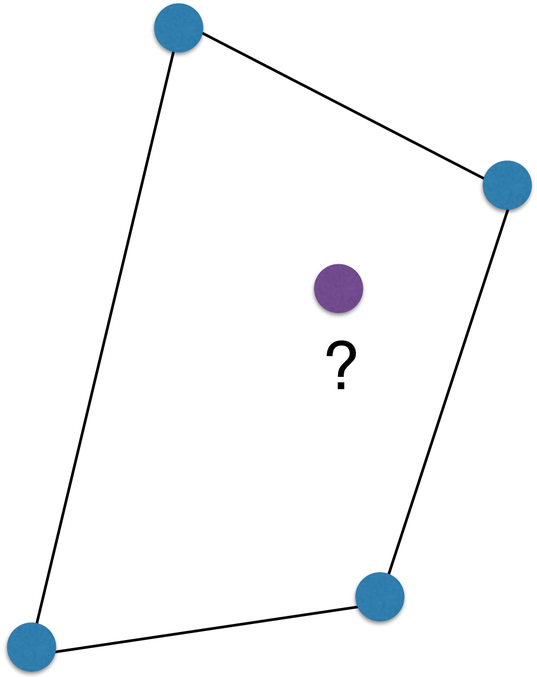
Bicubic

- Same idea as bilinear but using a 4 x 4 grid of neighbors
- Tends to produce sharper results
- More computationally intensive



Generalizing Bilinear

- Corner points don’t have to lie on a square or rectangle
- Can be any quadrilateral



Generalized Bilinear

- General form:

$$f(x, y) = ax + by + cxy + d$$

- Use similar strategy as with curve fitting:

$$\begin{aligned} f(x_1, y_1) &= ax_1 + by_1 + cx_1y_1 + d \\ f(x_2, y_2) &= ax_2 + by_2 + cx_2y_2 + d \\ f(x_3, y_3) &= ax_3 + by_3 + cx_3y_3 + d \\ f(x_4, y_4) &= ax_4 + by_4 + cx_4y_4 + d \end{aligned}$$

Solve for a, b, c, d and plug into function

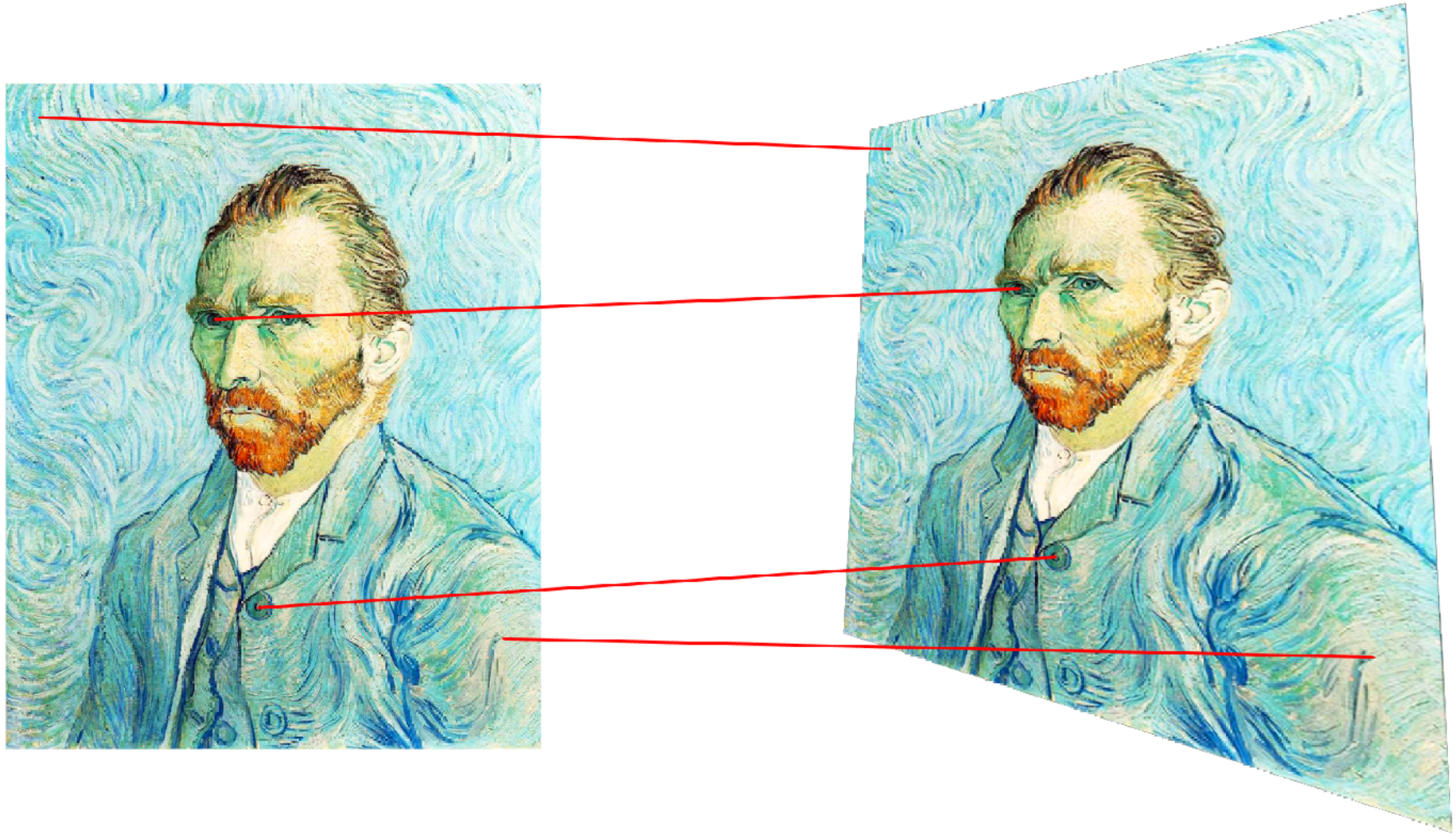
Coming up…

- Image warping
- Texture mapping

Image Warping (cont'd)

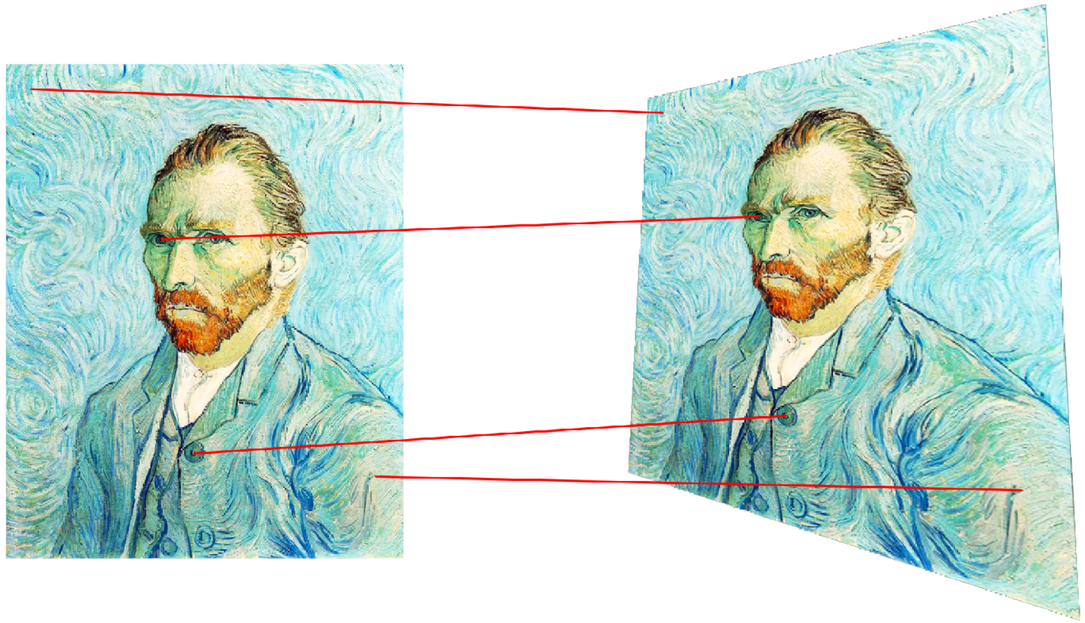
CS 355: Introduction to Graphics and Image Processing

Perpective Warping



Perpective Warping

- You've already seen how to project 3D geometry to create an image
- Special case: if the points all lie on a plane in 3D, the transformation can be done with a single 3 x 3 matrix
- Perspective projection of any 3D plane to any other 3D plane

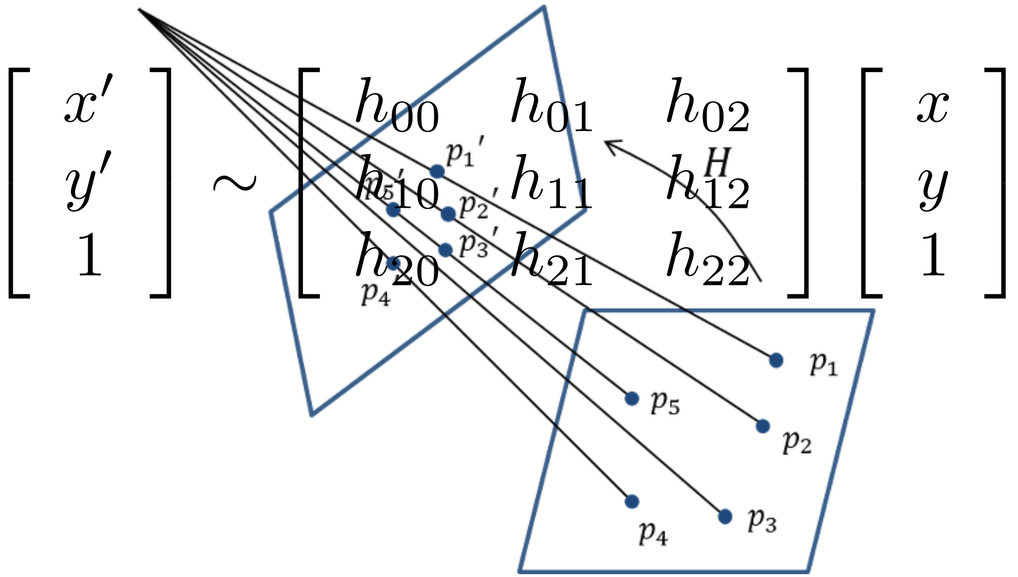


Homographies

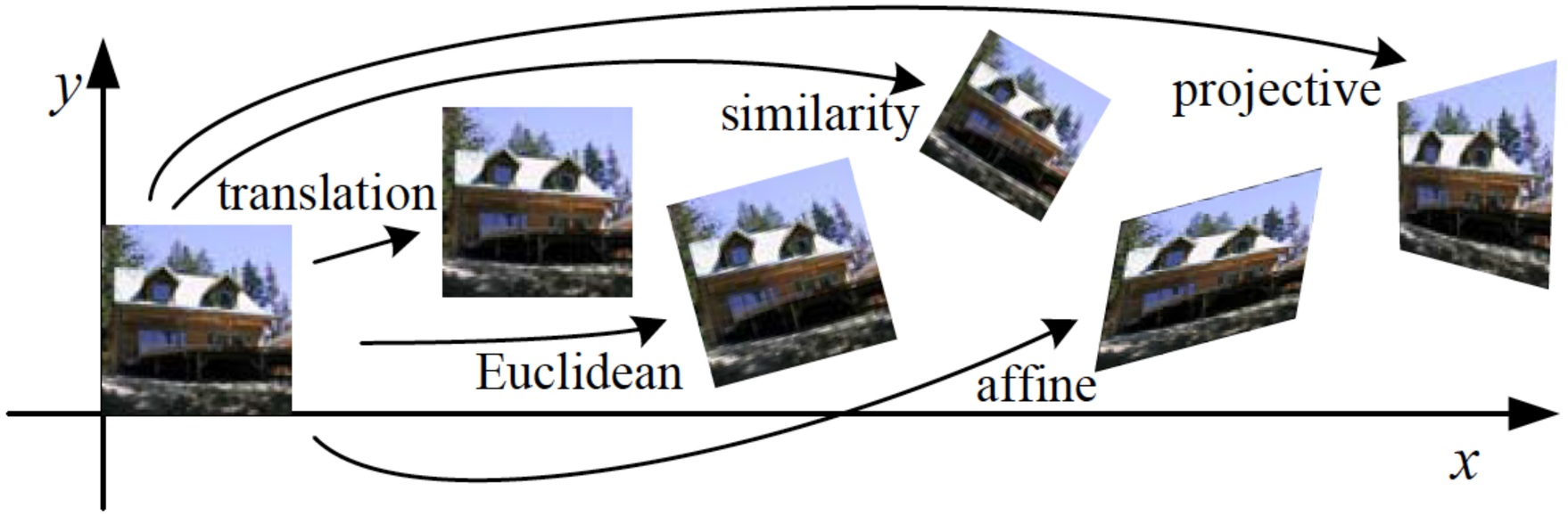
- Perspective projection from a plane to another plane is a homography
- Same 3 x 3 matrix transformation you've seen before, but the bottom row doesn't have to be [0 0 1]
- In practice, it's invertible!

$$\mathbf{p}' = \mathbf{H}\ \mathbf{p}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Classes of Transformations



Homographies

- Non-zero multiple of a homography does the same transformation
- That means there are really only 8 free parameters, not 9

$$\mathbf{p}' = \mathbf{H}\ \mathbf{p}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Multiple Pictures of the Same Plane

- If two cameras take pictures of a plane from different viewpoints,
  - The projection of the plane to each camera is a homography
  - The transformation of points from one picture to the other is also a homography!

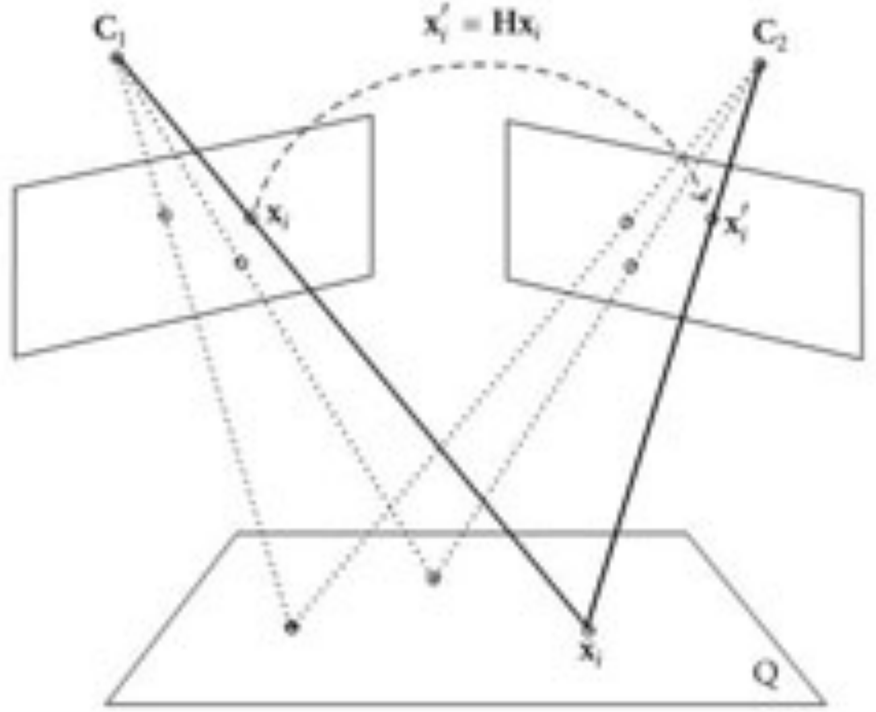


Image Stitching

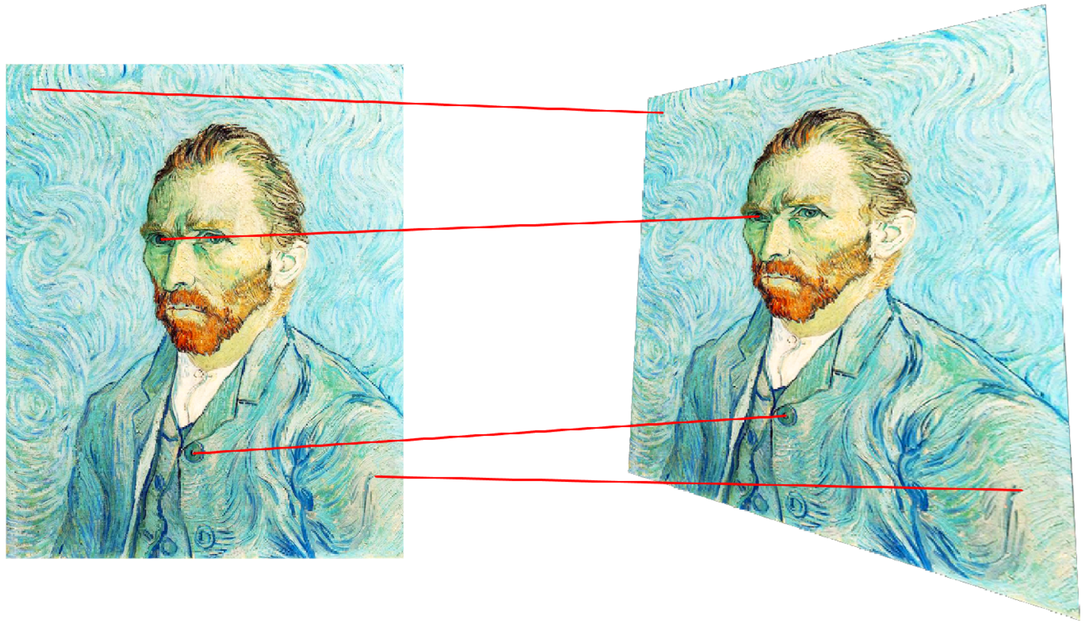
“Straight On” View

“Birds-Eye” View

Or Going The Other Way

Computing Homographies

- Eight free parameters
- A pair of matching points provides two constraints (one for x, one for y)
- Four points provides eight equations to solve for eight unknowns
- Call the *4-point algorithm*



Lab 9

Lab 9

Lab 9

- We give you an implementation of the 4-point algorithm:
  - You give it matching points
  - It gives you back a homography
- You implement:
  - Backward warping
  - Bilinear interpolation
  - Point-in-frame geometric tests (more on this later)

Coming up...

- Back to more geometry:
  - Representing lines, curves, surfaces
  - Geometric tests

Lines, Curves, and Surfaces

CS 355: Introduction to Graphics and Image Processing

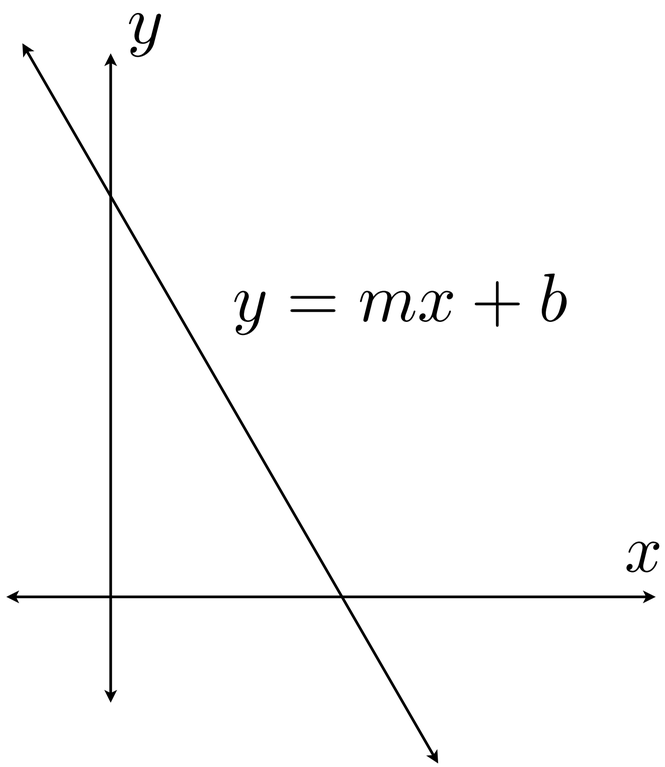
Lines

Lines extend infinitely in both directions

Line segments are finite

Rays extend infinitely in one direction

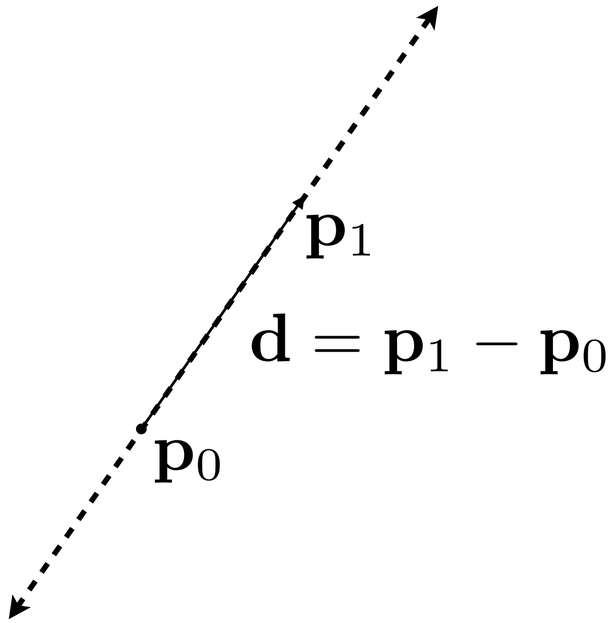
Slope / Intercept



Doesn’t work for vertical lines

Hard to extend to 3D

Parametric Representation



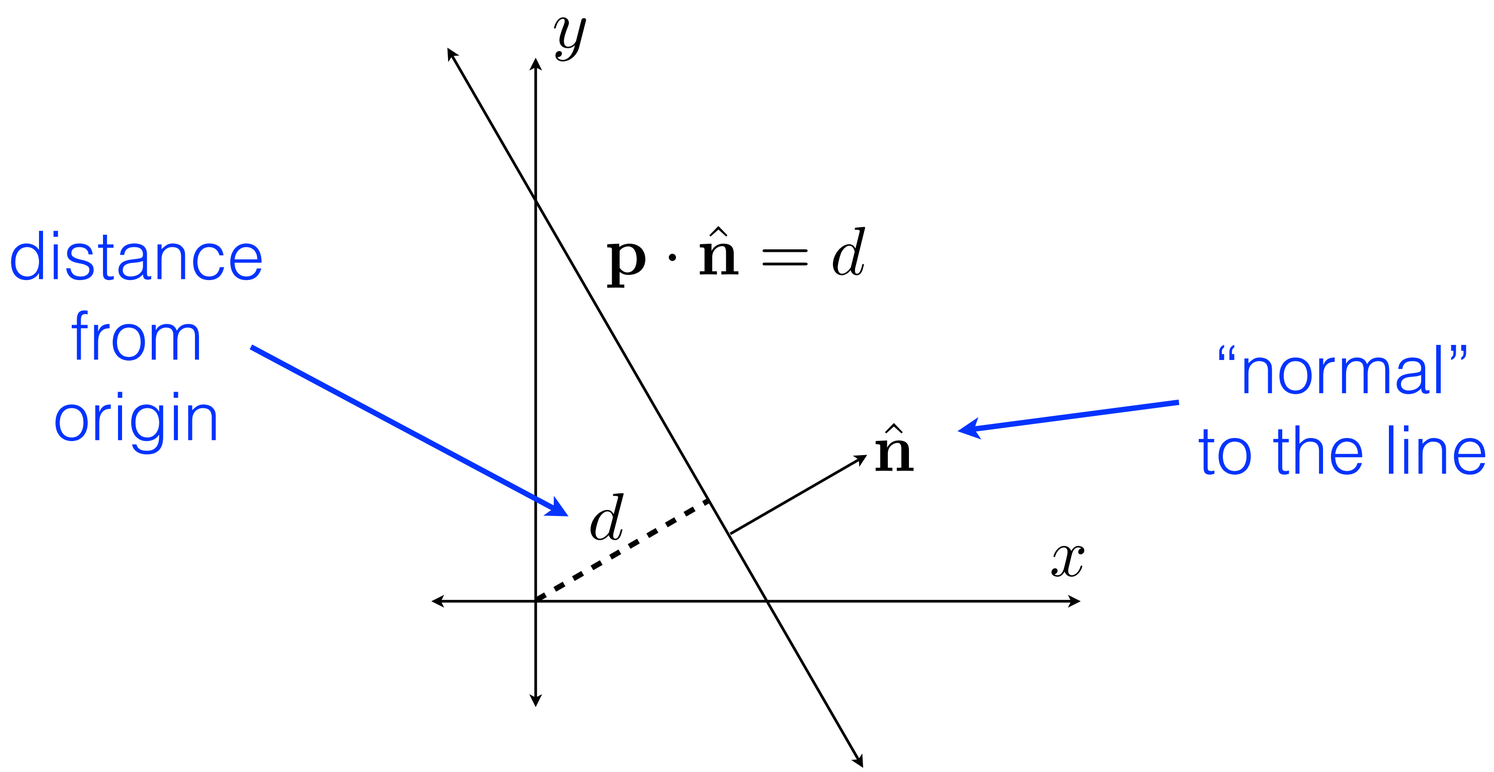
$$\mathbf{p}_0 + t\ \mathbf{d}$$

Line: $-\infty < t < \infty$

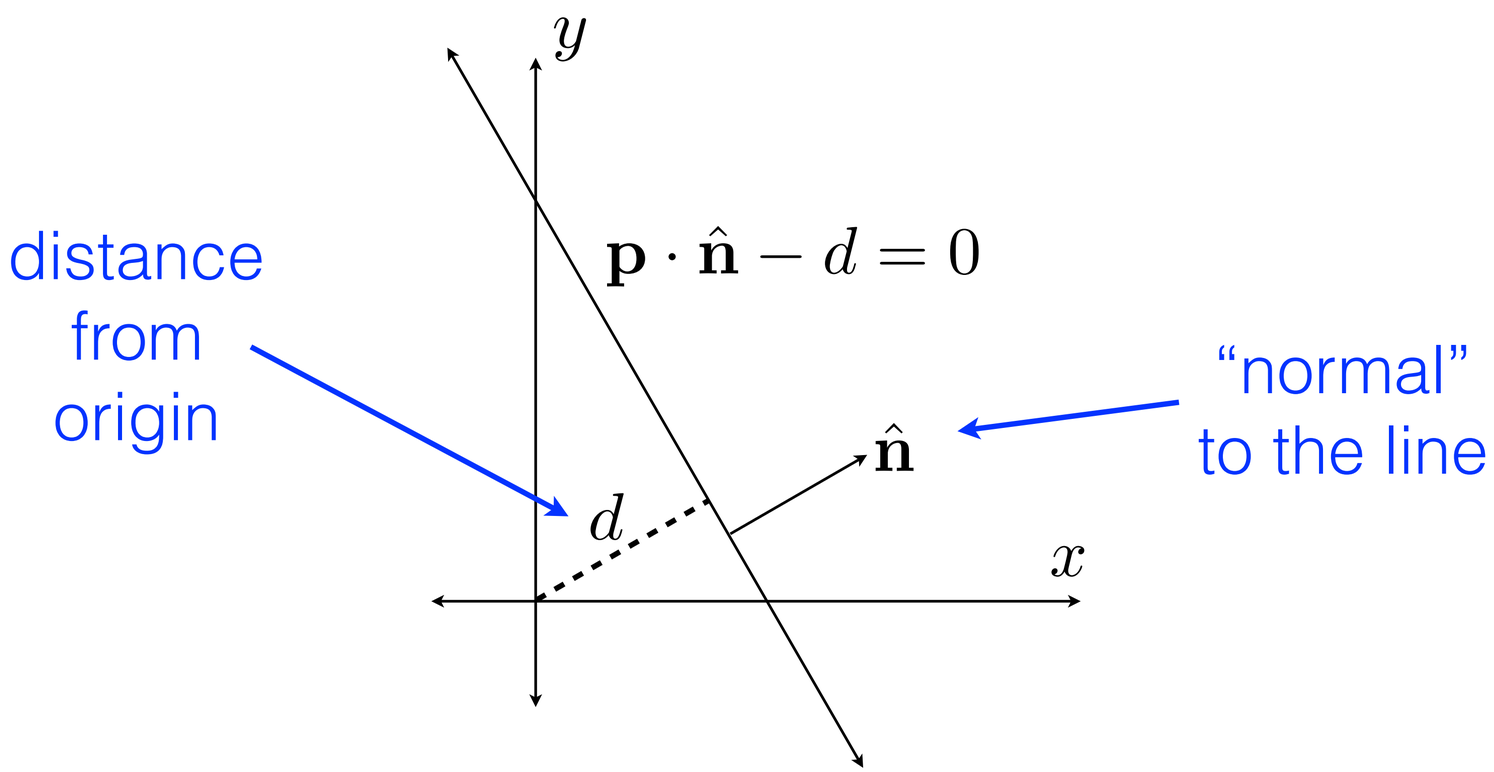
Line segment: $0 \leq t \leq 1$

Ray: $0 \leq t < \infty$

Normal + Distance



Implicit Representation



Aside: Lines and Distance

You've seen this before:

$$ax + by + c = 0$$

normal

negative distance

Normal and distance:

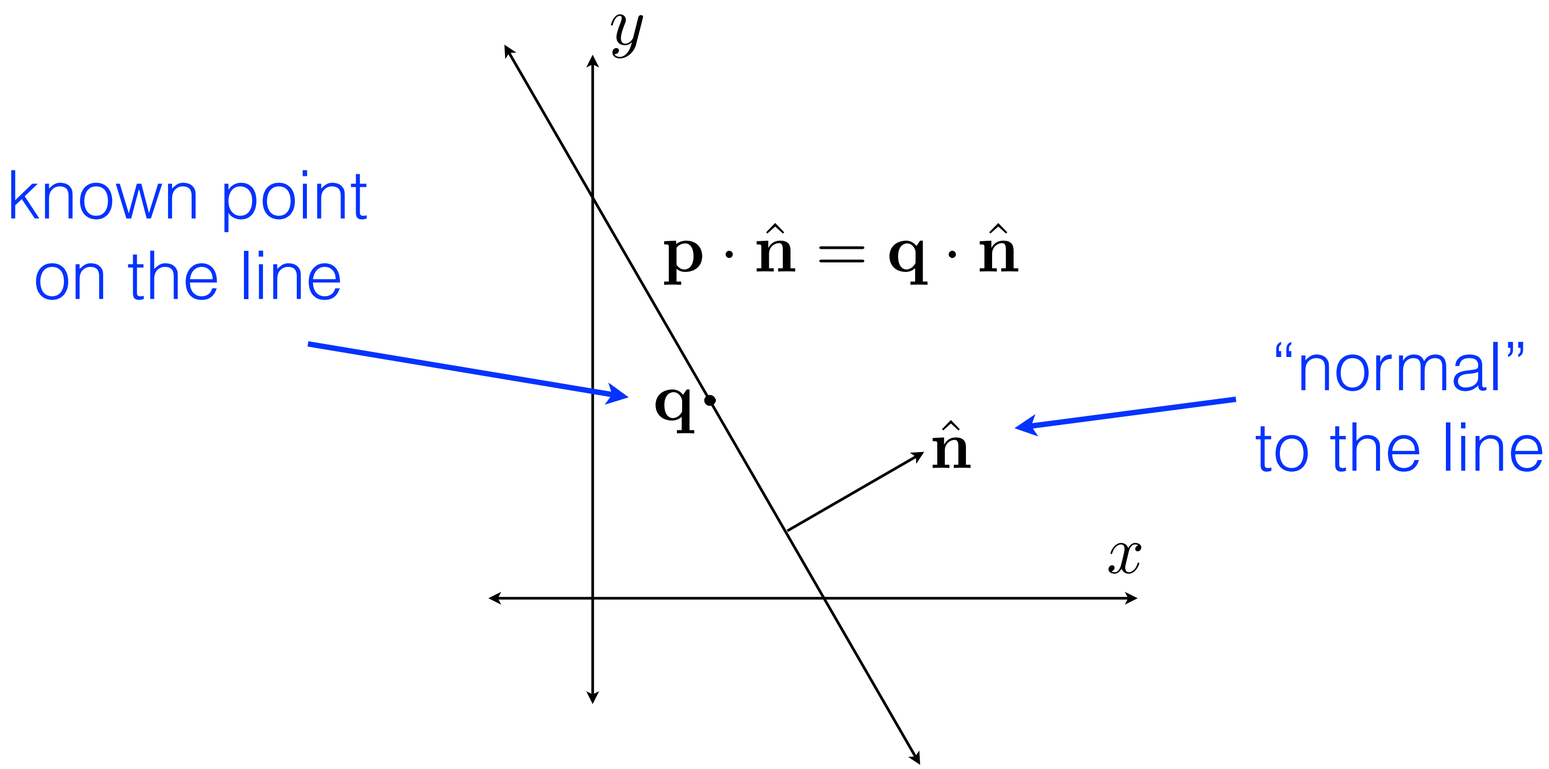
$$\mathbf{p} \cdot \hat{\mathbf{n}} = d$$

$$\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \hat{\mathbf{n}} = \begin{bmatrix} a \\ b \end{bmatrix}$$

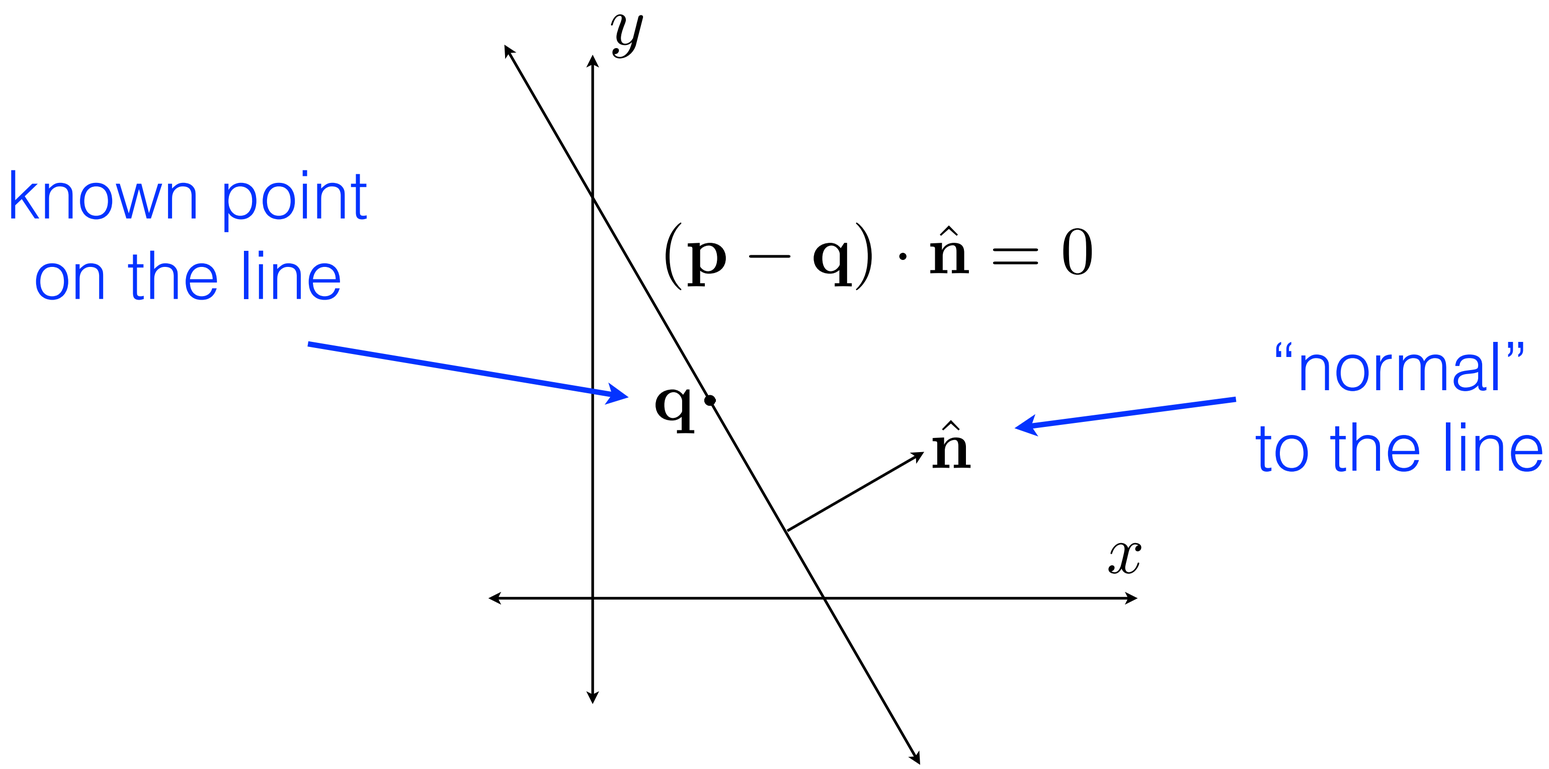
$$d = -c$$

This is really what you've seen years ago,
just in linear algebra form

Normal + Point



Implicit Representation



Representing Shapes

- Parametric
  (sweeps out the shape as a function of some parameters)

$$\mathbf{p}(t)$$

- Implicit
  (meets some test)

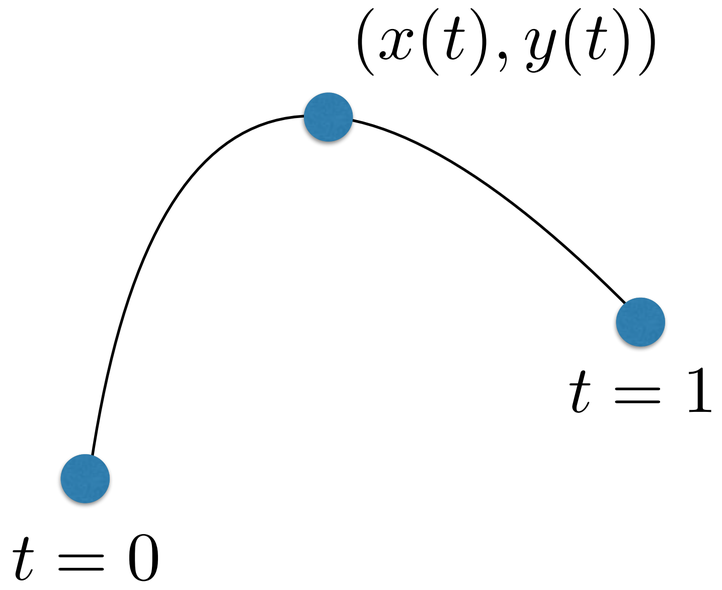
$$f(\mathbf{p}) = 0$$

- Others...

Can usually convert between representations

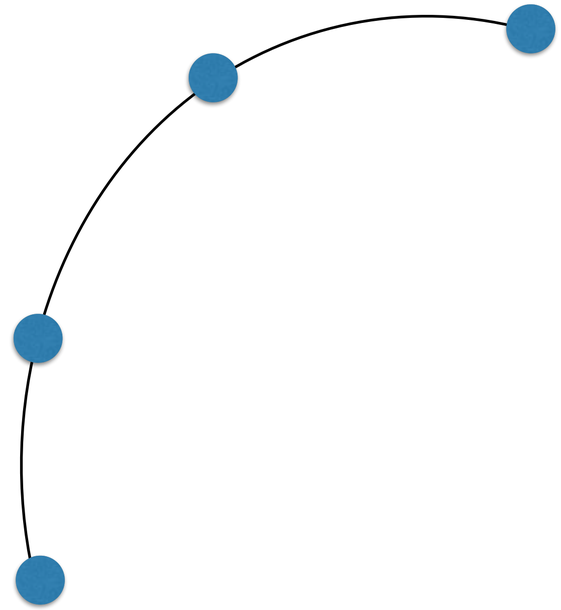
2D Curves

- Parametric curves
  - Parameter *t* traces the curve
  - One function for each dimension
  - Can extend to a space curve in any 3D or higher



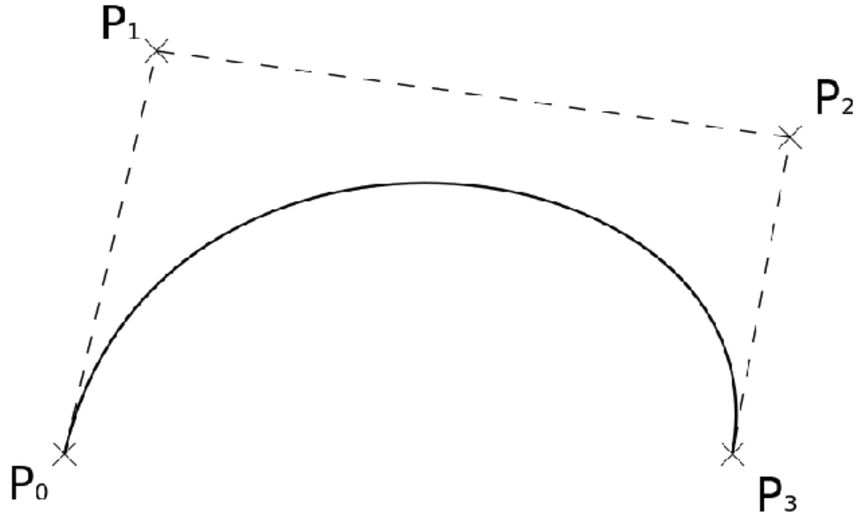
Cubic Curves

- Lots of types
  - Exact-fit curves (splines, etc.)
  - Bezier
  - B-splines
  - …



Bezier Curves

- Specify *end points* and a set of *control points* in between
- Curve doesn't (usually) pass through the control points
- Control points determine tangents at certain points on the curve
- Common in 2D drawing programs

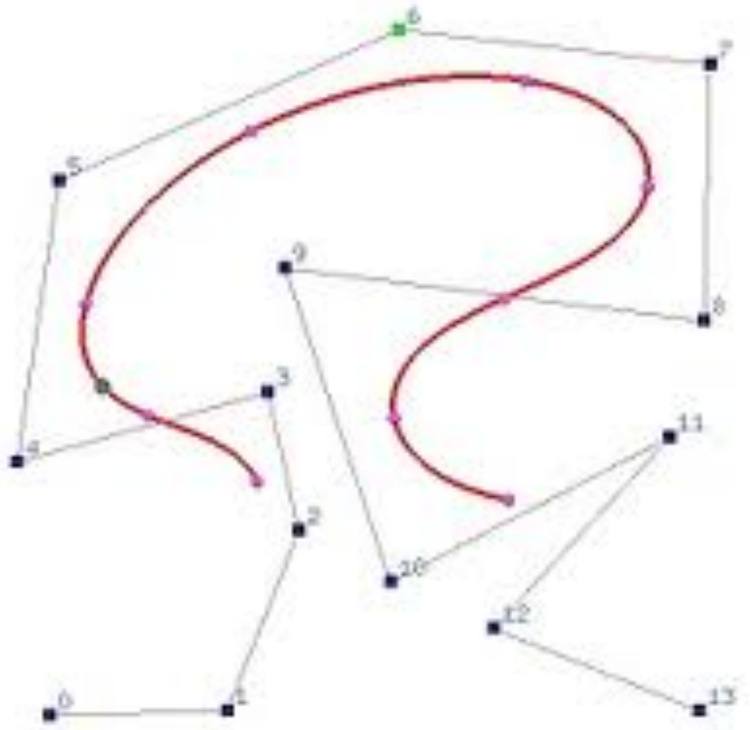


Complicated Curves

- Use higher-degree polynomials
  - Can make an nth-order Bezier curve out of two end points and *n-1* control points
  - Cumbersome and slow for long curves
  - Non-local control!
- Piecewise cubic segments
  - Can get 0th-order continuity by sharing endpoints
  - Can get 1st-order continuity by aligning control points

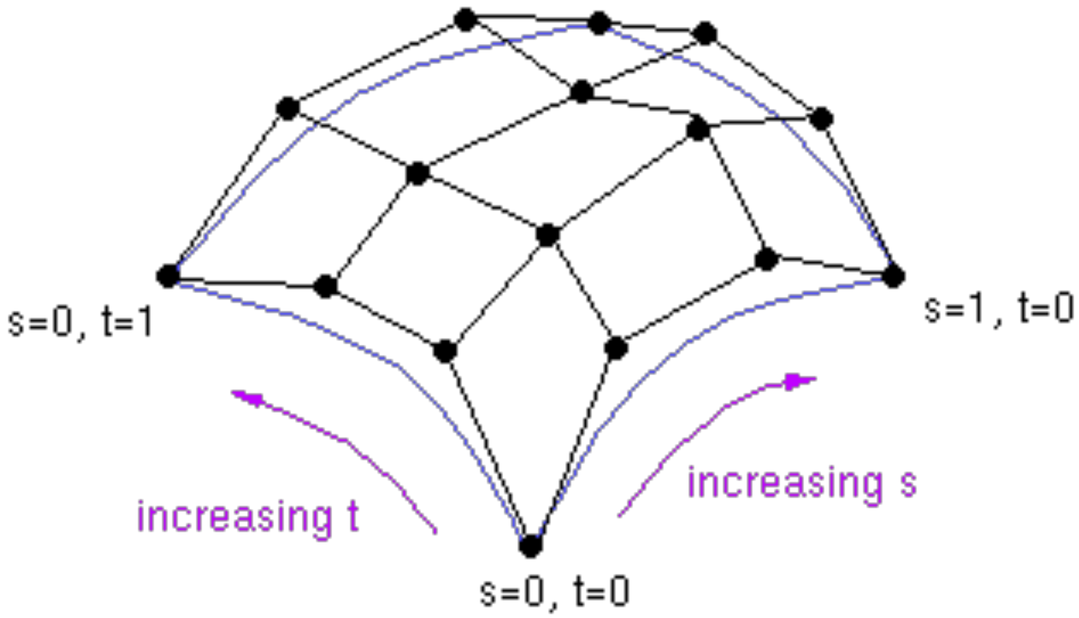
B-Splines

- More common to use *basis splines* (B-splines)
- Each local portion of the curve is a *weighted blend* of nth-order basis functions (usually cubic)



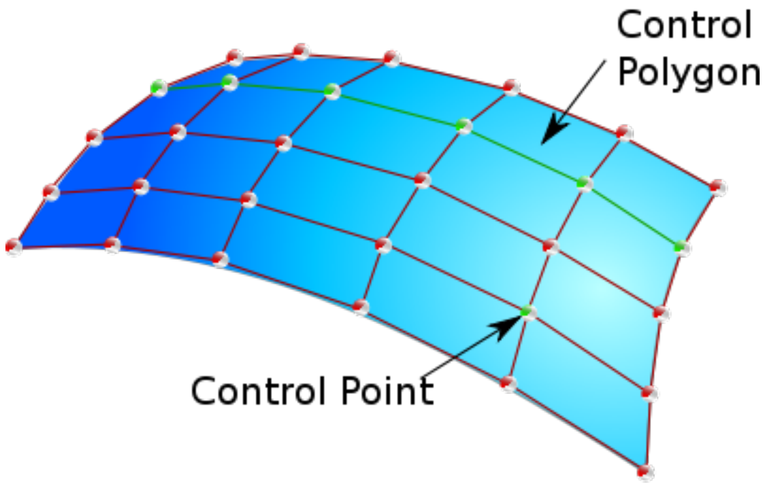
3D Surfaces

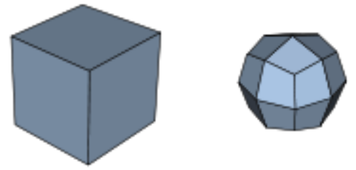
- Extend idea of Bezier or B-spline curves to a mesh or grid of control points
- Two parameters across surface: s, t
- Similar in principle to bicubic interpolation



3D Surfaces

- Bezier
- B-spline
- Non-Uniform Rational B-splines (NURBs)
- Catmull-Rom and other spline variations
- Subdivision surfaces





See CS 455 for more…

Coming up...

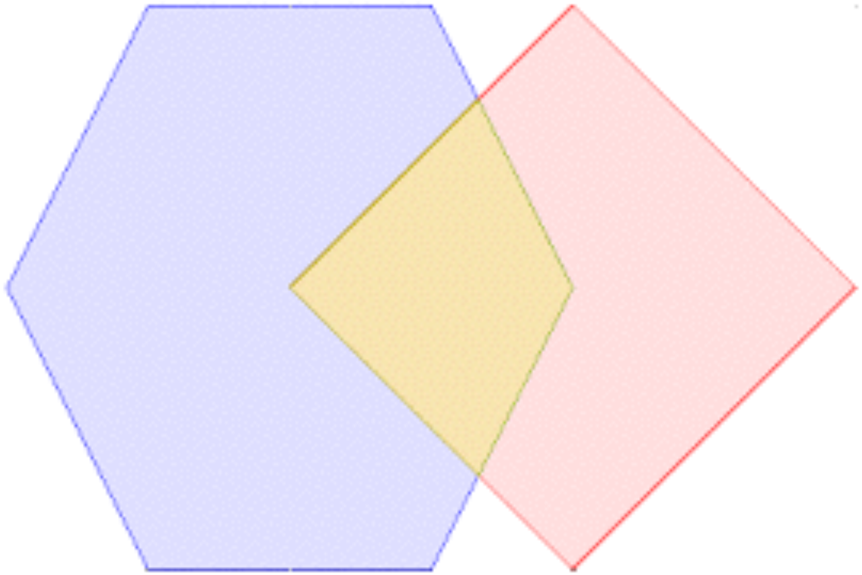
- Geometric tests

Geometric Tests

CS 355: Introduction to Graphics and Image Processing

Geometric Tests

- Lots of things in graphics involve making geometric tests:
  - Interactive selection
  - Intersections
  - Collisions (e.g., games)
  - Ray intersections
  - Nearest points
  - …



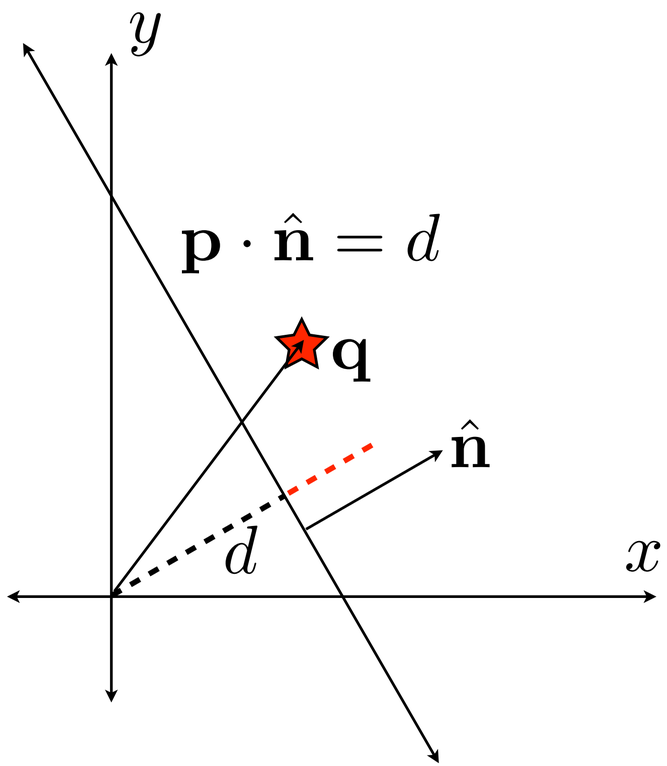
Distance to a Line

- Points **p** on the line **L** satisfy this constraint:

$$\mathbf{p} \cdot \hat{\mathbf{n}} - d = 0$$

- Distance from point **q** to the line **L**:

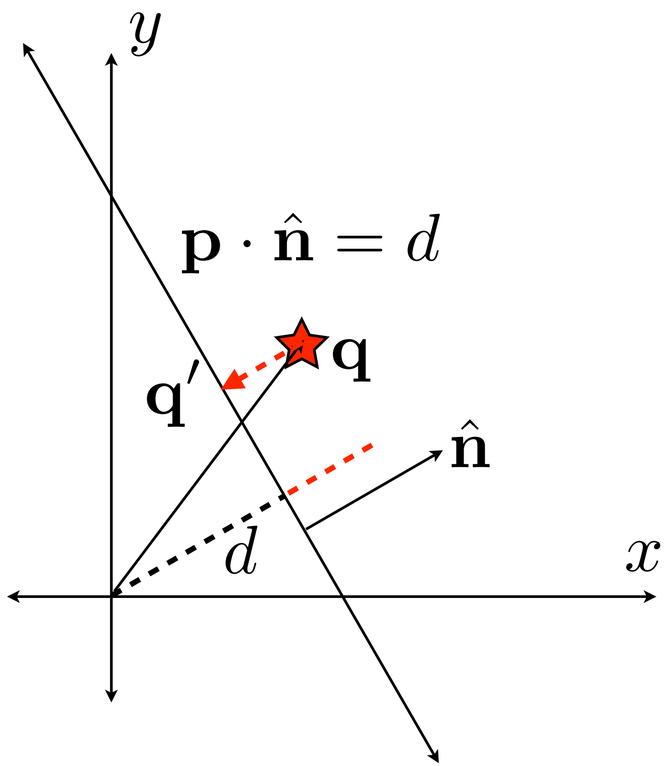
$$|\mathbf{q} \cdot \hat{\mathbf{n}} - d|$$



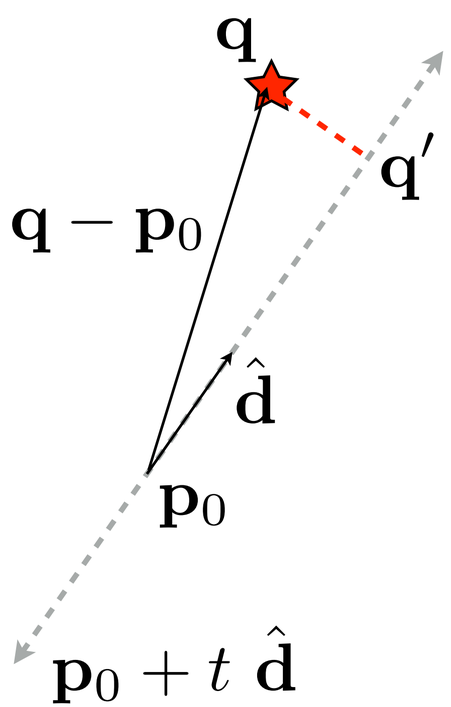
Closest Point to a Line

- To get the closest point on the line **L** to point **q**, go back along the normal direction:

$$\mathbf{q}' = \mathbf{q} - (\mathbf{q} \cdot \hat{\mathbf{n}} - d)\ \hat{\mathbf{n}}$$



Closest Point to a Line



At what value of t is
the line closest to q?

$$t = (\mathbf{q} - \mathbf{p}_0) \cdot \hat{\mathbf{d}}$$

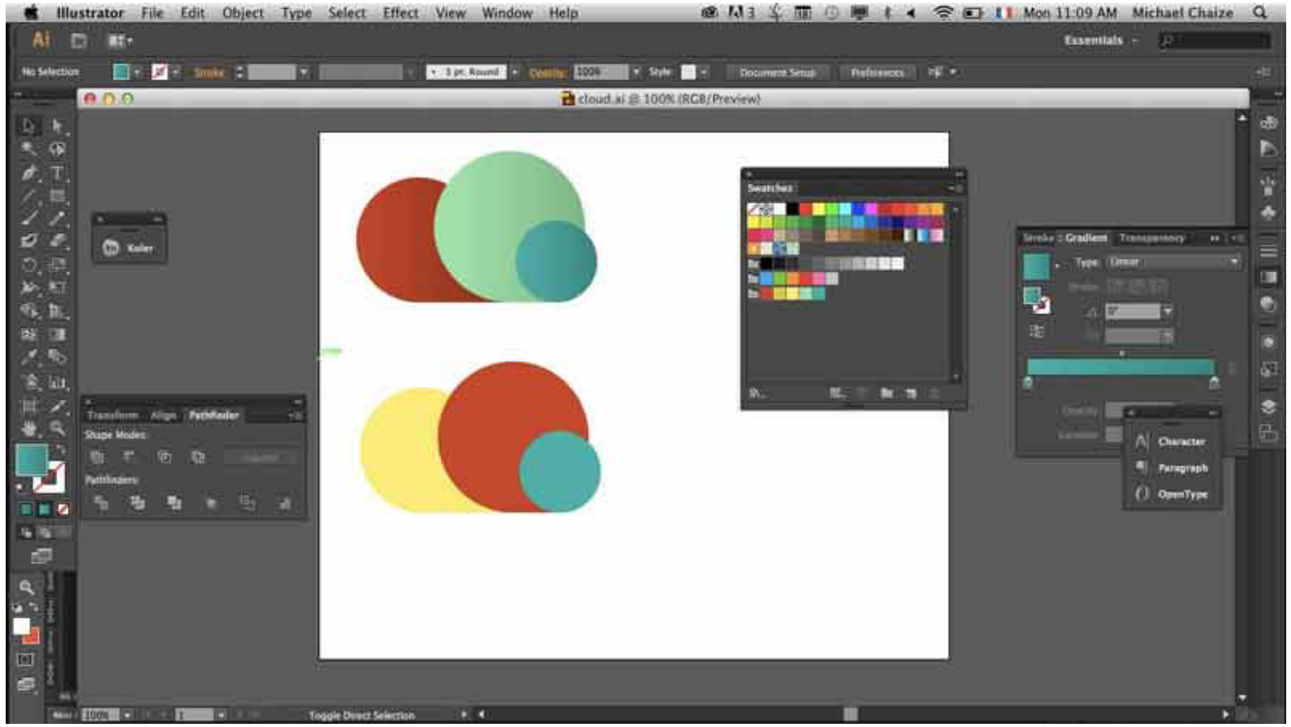
$$\mathbf{q}' = \mathbf{p}_0 + ((\mathbf{q} - \mathbf{p}_0) \cdot \hat{\mathbf{d}})\ \hat{\mathbf{d}}$$

$$\text{distance} = \|\mathbf{q} - \mathbf{q}'\|$$

Works for higher dimensions as well!

Point-In-Shape Tests

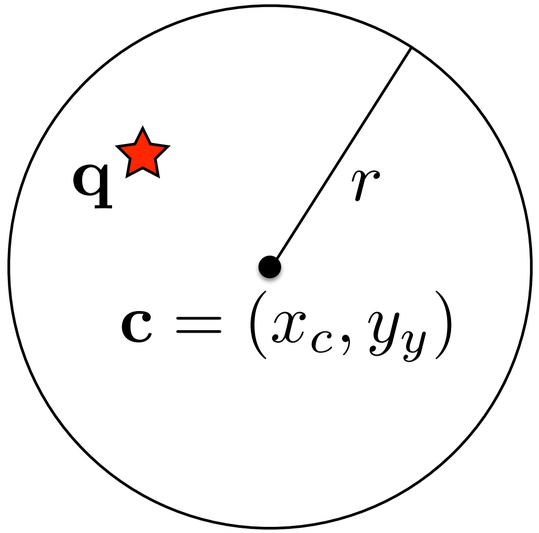
- Useful for selection tests in graphical interaction
- “Did the user click on…”



Point in Circle

Point on the circle:

$$\|\mathbf{q} - \mathbf{c}\| = r$$

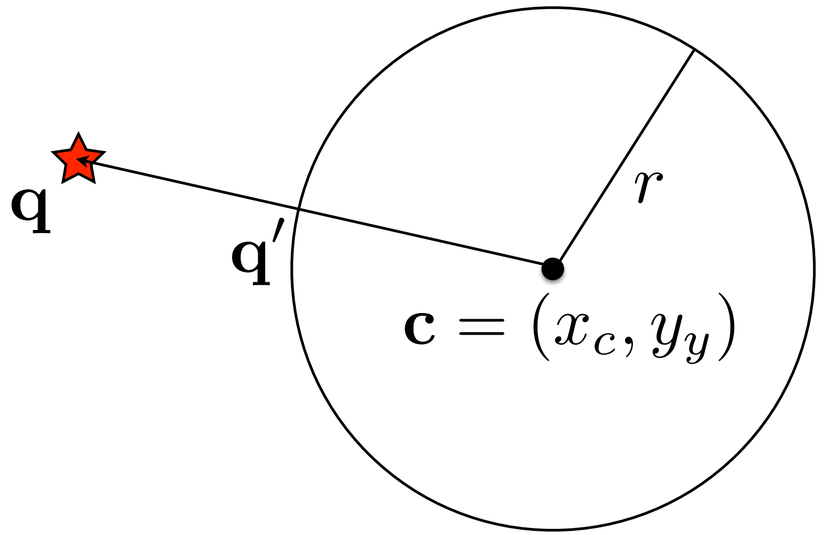


Point in the circle:

$$\|\mathbf{q} - \mathbf{c}\| \leq r$$

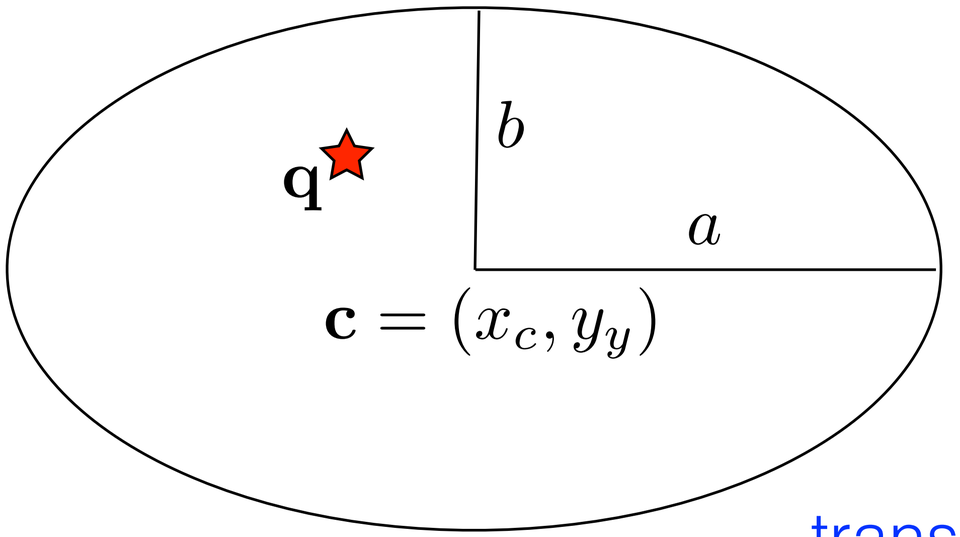
$$(q_x - c_x)^2 + (q_y - c_y)^2 \leq r^2$$

Closest Point on Circle



$$\mathbf{q}' = \mathbf{c} + r \, \frac{\mathbf{q} - \mathbf{c}}{\|\mathbf{q} - \mathbf{c}\|}$$

Point in Ellipse (Oval)



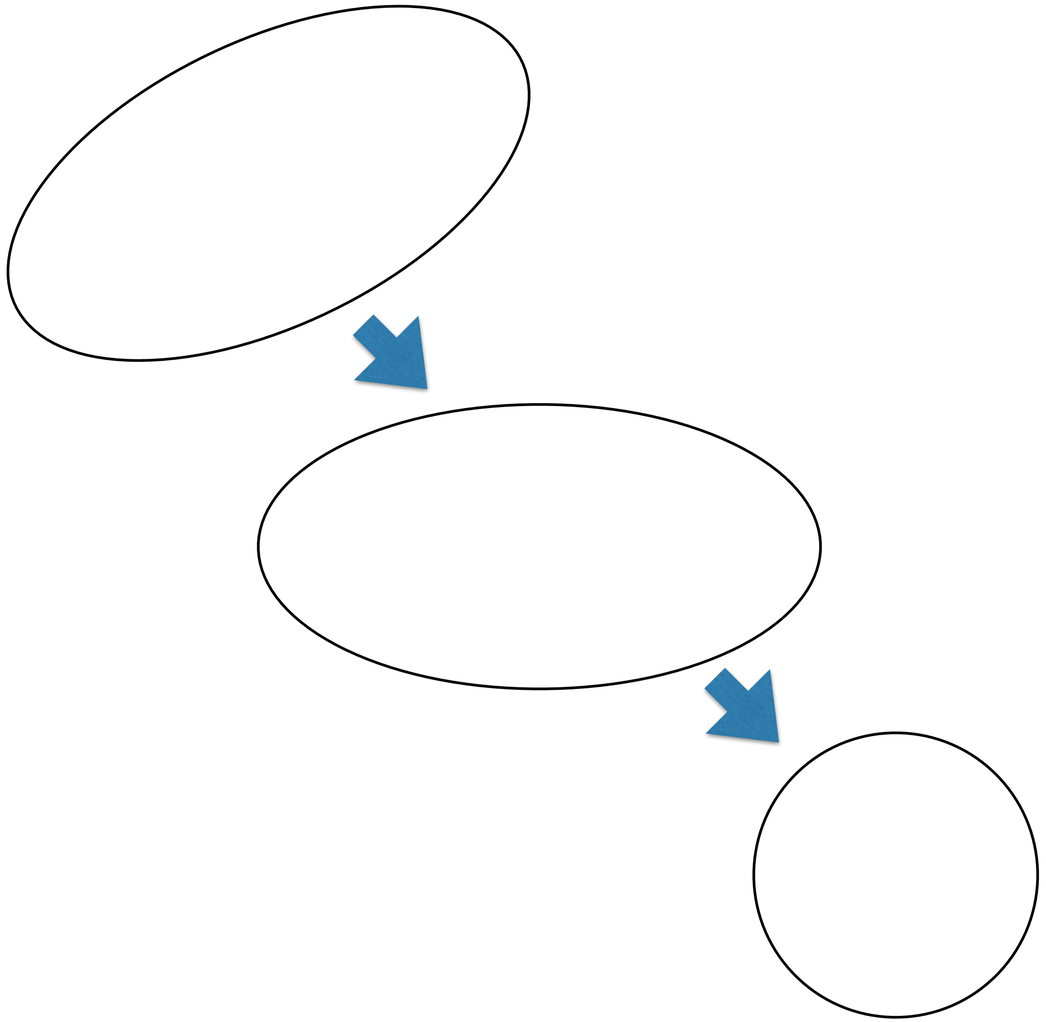
translate!

$$\left(\frac{q_x - c_x}{a}\right)^2 + \left(\frac{q_y - c_y}{b}\right)^2 \leq 1$$

scale!

Transformations and Tests

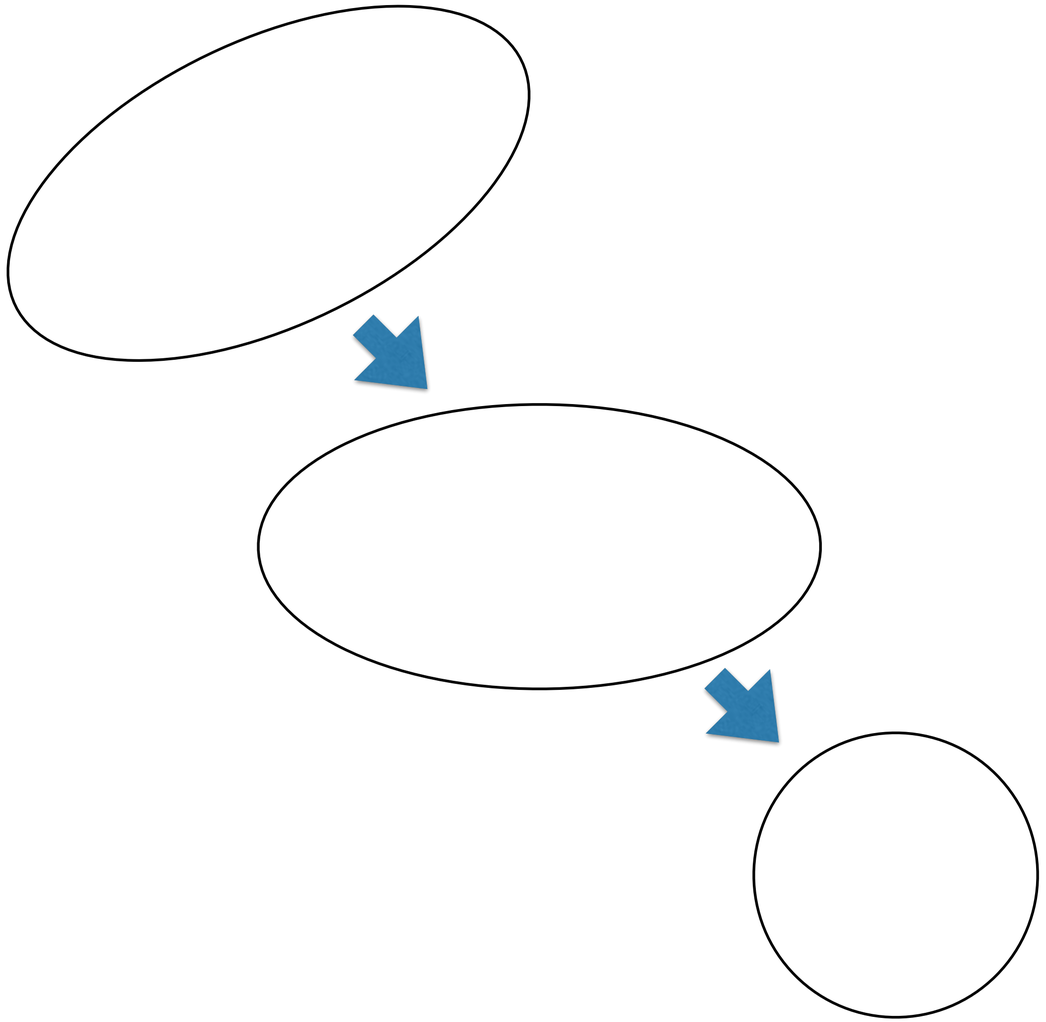
- What seem like complicated tests become easier after transformation
- Center the shape
- Rotate to standard alignment
- Scale if necessary
- Simple tests



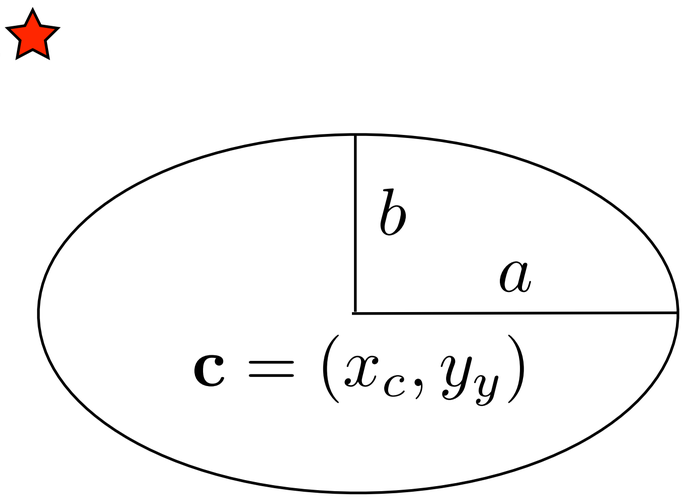
Transformations and Tests

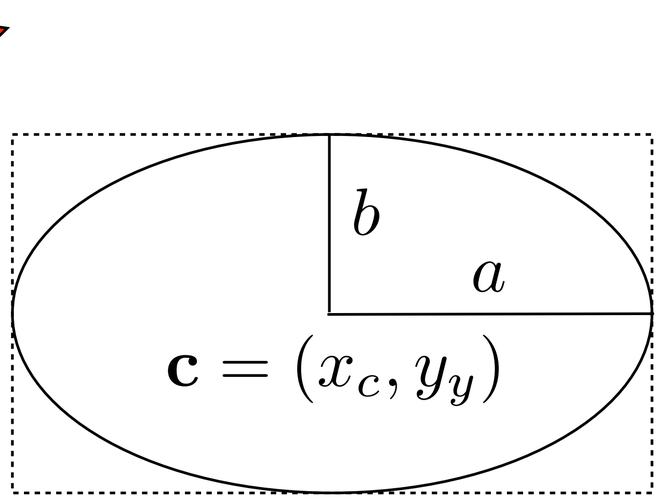
- **Object-to-world** for rendering

- **World-to-object** for selection (geometric tests are often easiest the object's own intrinsic space)



Bounding Boxes





Hard test:

$$\left(\frac{q_x - c_x}{a}\right)^2 + \left(\frac{q_y - c_y}{b}\right)^2 \leq 1$$

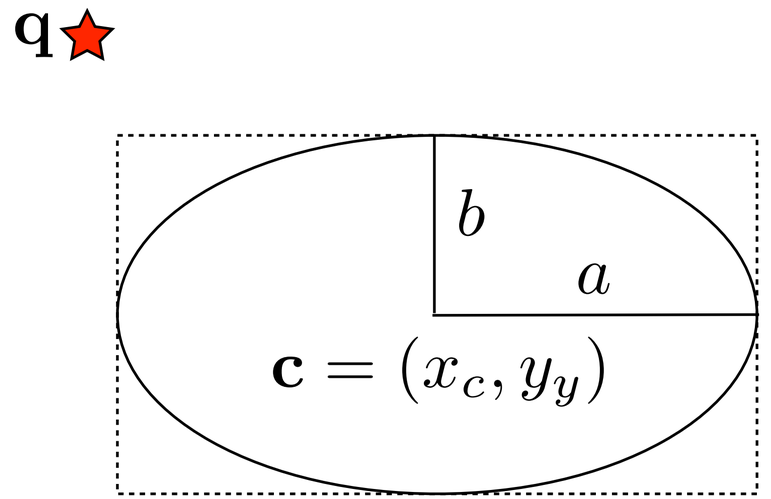
Easy test:

$$|q_x - c_x| \leq a$$

$$|q_y - c_y| \leq b$$

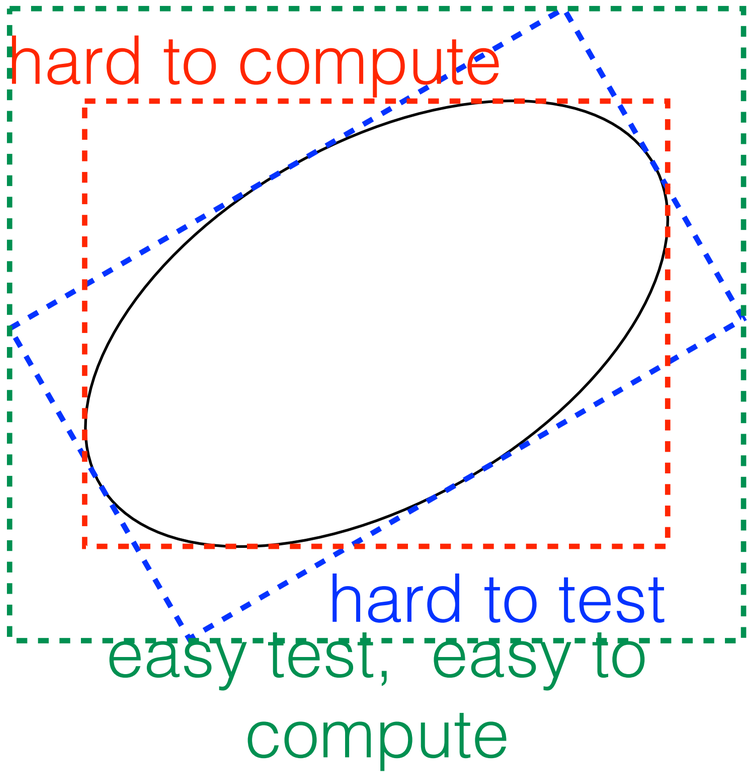
Bounding Boxes

- Idea: use bounding box tests as a “quick reject”
- If passes, then spend time on more complex tests
- The more complex the test, the bigger the win



Bounding Boxes

- Boxes don’t necessarily have to be “tight” to be useful
- Looser boxes may be easier to compute and test
- “Axis-Aligned Bounding Box” (AABB)

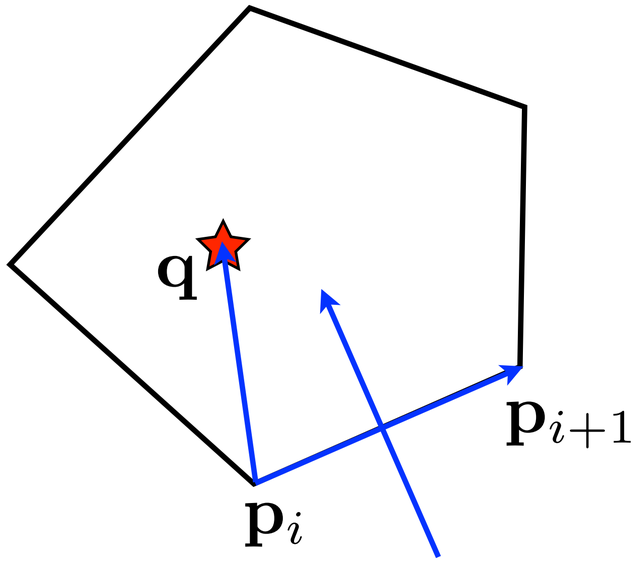


Convex Polygons

- In 2D: for all edges, the point is on the same side of the edge
- Walk around the polygon (in order) and test

$$(\mathbf{q} - \mathbf{p}_i) \cdot (\mathbf{p}_{i+1} - \mathbf{p}_i)_\perp > 0$$

- Be consistent with ordering and perpendiculars

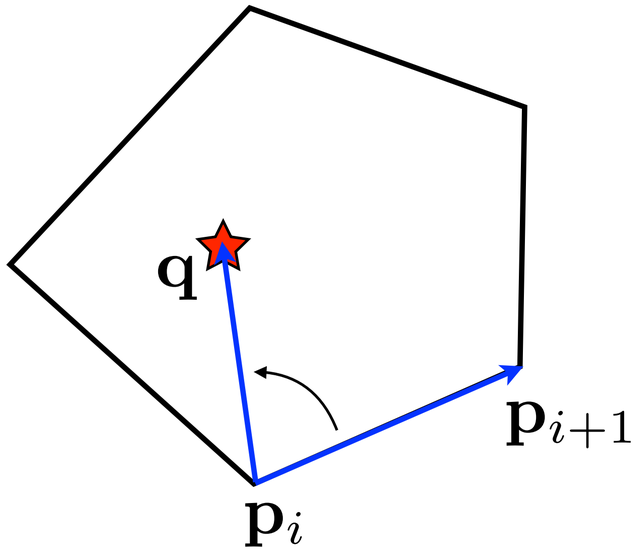


Convex Polygons

- Can also do using cross-products
- Turn all 2-D points (x,y) to 3-D ones (x,y,0)
- Test the z component of this cross-product:

$$(\mathbf{p}_{i+1} - \mathbf{p}_i) \times (\mathbf{q} - \mathbf{p}_i)$$

- If all are positive, the point is in the shape



Polygon Bounding Boxes

- Bounding boxes for polygons are really easy
- Just min / max tests over all of the vertices' x and y coordinates

$$\min(x_i)$$

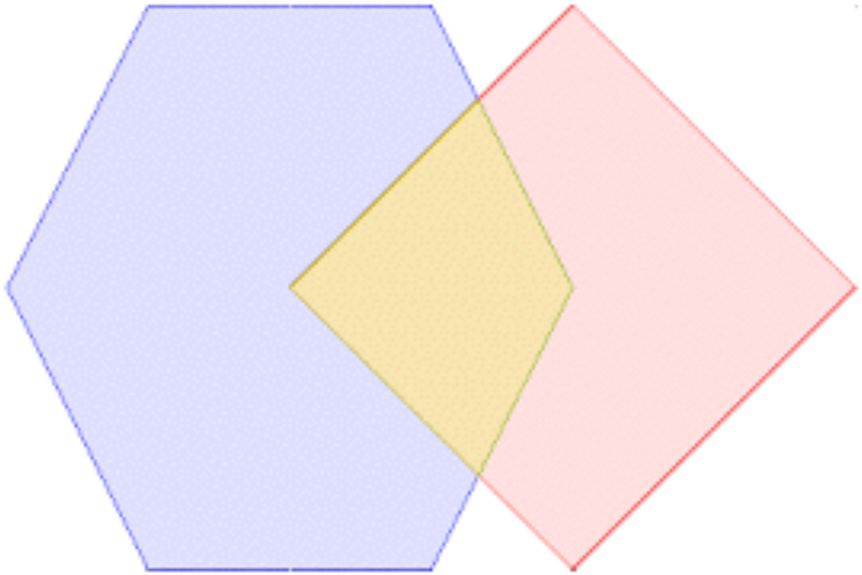
$$\min(y_i)$$

$$\max(x_i)$$

$$\max(y_i)$$

Intersecting Polygons

- To test to see if two convex polygons intersection, see if any of the vertices of one are within the other, and vice versa



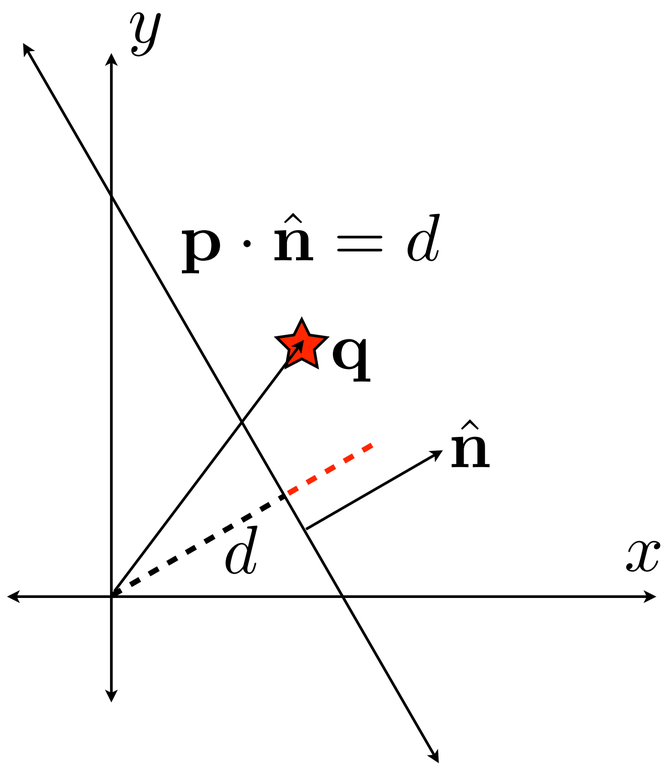
Which Side of a Line?

- Points **p** on the line **L** satisfy this constraint:

$$\mathbf{p} \cdot \hat{\mathbf{n}} - d = 0$$

- Use to test which side of a line a point is on by looking at the sign of

$$\mathbf{q} \cdot \hat{\mathbf{n}} - d$$



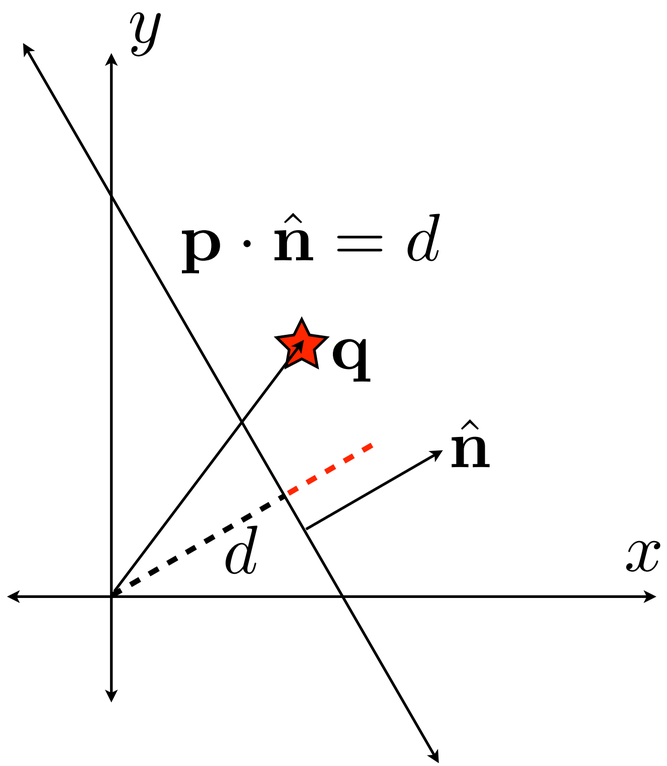
Which Side of a Plane?

- A 2-D plane in 3-D space can be represented the same way:

$$\mathbf{p} \cdot \hat{\mathbf{n}} - d = 0$$

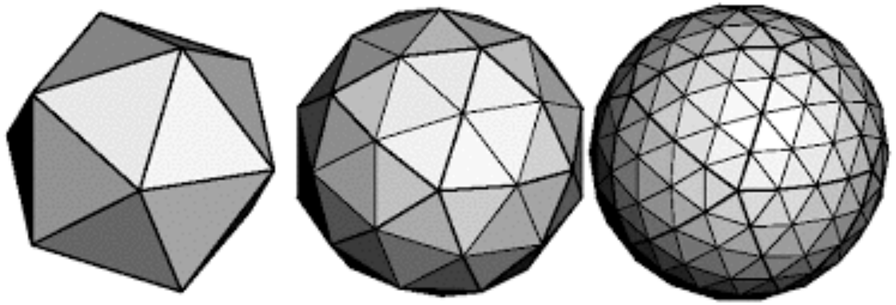
- Use to test which side of a plane a point is on by looking at the sign of

$$\mathbf{q} \cdot \hat{\mathbf{n}} - d$$



Point Inside Polygonal Model

- A point is inside a convex polygonal model if it is on the “inside” side of all of the faces
- Can test for intersection of two convex shapes by seeing if a vertex of another is inside the other, and vice versa



Lab 9

- Don’t perform backward mapping for every point in the image
- Compute a bounding box around the four corners of the frame (min/max on the x, y coordinates)
- Test to see if each point is inside the quadrilateral of the frame (point in convex polygon test)

Lab 9

- We give you an implementation of the 4-point algorithm:
  - You give it matching points (match corners of the target frame to corners of the image)
  - It gives you back a homography
- You implement:
  - Point-in-frame geometric tests
    - Iterate over bounding box
    - Point-in-quadrilateral tests
  - Backward warping
  - Bilinear interpolation

Coming up...

- Frequency-domain processing

Introduction to the Frequency Domain

CS 355: Introduction to Graphics and Image Processing

Let’s revisit the idea of transformations…

Basis Sets

- Remember that a basis set is a minimal set of vectors that span a space of vectors
- That means any vector in the space can be represented by a unique sum of the basis vectors
- **All vectors are represented with respect to some basis set**

$$\{\mathbf{e}_i\}$$

$$\mathbf{v} = \sum_i a_i \ \mathbf{e}_i$$

Change of Basis

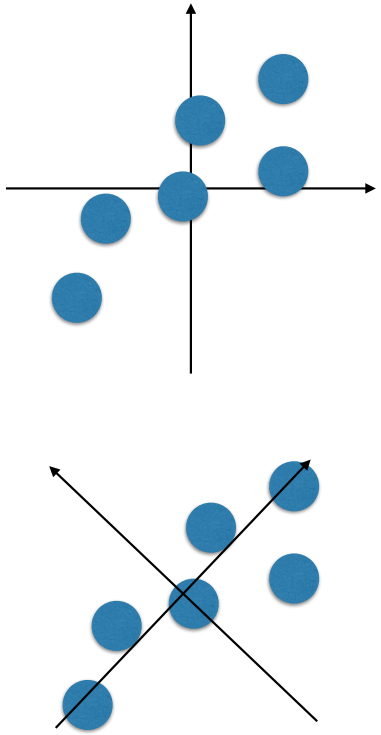
- Can change from
  - representation in terms of one basis set to
  - representation in terms of another basis set
- If basis vectors are orthonormal, this is just simple dot products (you've seen this already)

$$\mathbf{v} = \sum_i a_i \ \mathbf{e}_i$$

$$a_i = \mathbf{v} \cdot \mathbf{e}_i$$

Change of Basis

- For many problems, *analyzing points and vectors may be easier in a different coordinate system*
- Key is often to find the right coordinate system
  - Can be based on the problem
  - Can adapt to the specific data

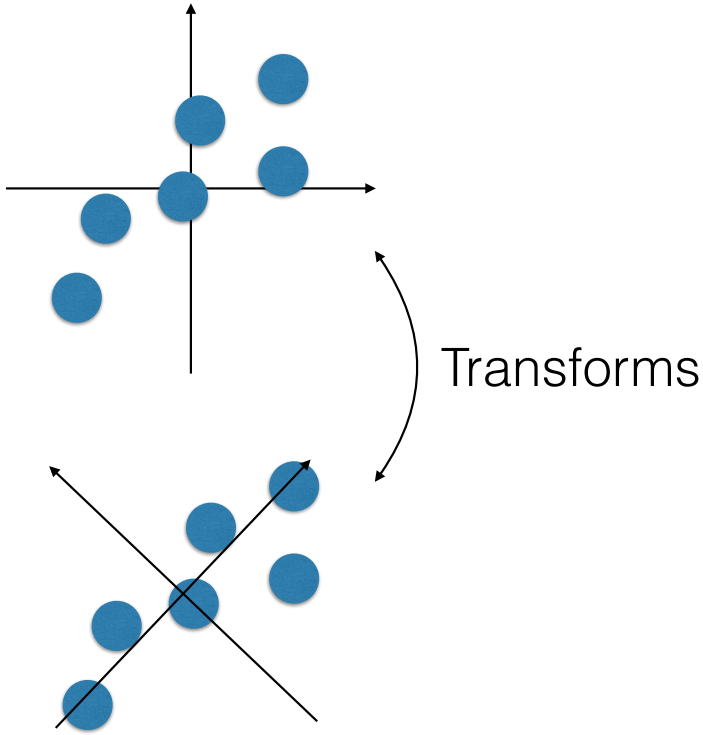


Data as Vectors

Lots of things can be thought of as points/vectors in some space

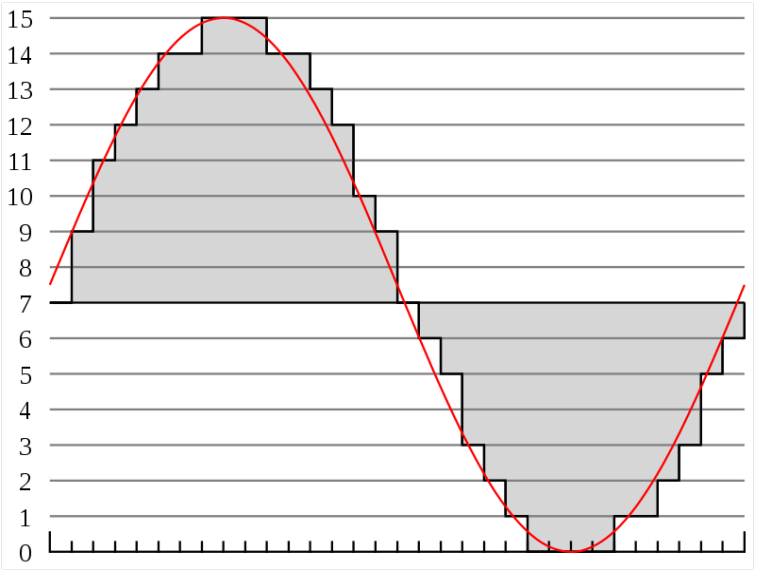
Data Transforms

- Common pattern in data analysis:
  - Represent data as vectors
  - Convert to a different coordinate system
  - Analyze (or change!) while in that coordinate system
  - Convert back (if needed)



Digital Audio

- Raw digital audio is stored as a series of sampled values (Pulse Code Modulation)
- One second of (one channel of) music on a CD is 44,100 samples
- Is this any different from a vector in a 44,100 dimensional space?
- Can we represent it differently?



Fourier Analysis

- Sampled sines and cosines of different frequencies form an orthonormal basis set
- Can decompose any waveform into a weighted sum of sines and cosines of different frequencies
- Great for analysis, manipulation, etc.

$$c[u] = \sum_k f[k] \ \cos(2\pi u t)$$

$$s[u] = \sum_k f[k] \ \sin(2\pi u t)$$

(OK, there's a bit more to it, but this is the basic idea.)

Let’s hear it…

Key Ideas from Today

- Transformations used for geometry extend to higher-dimensional data
- Can represent any point/vector as a weighted sum of basis vectors
- Choose the right set of basis vectors for your problem:
  - If the basis is orthonormal, get the weights by projecting (dot product)
  - Get back the original using a weighted sum of the basis vectors
- Sines and cosines are orthonormal basis set for 1-D functions like sound
- Can represent sound in the **time domain** or the **frequency domain**
- Can be used for analysis, but can also use it to manipulate the sound!

Coming up...

- A bit of review on complex numbers and sinusoids
- The Fourier Transform
- The Fast Fourier Transform
- How to feed data into FFT code and how to interpret the results
- Filtering in the frequency domain
- Sampling revisited
- Image filtering

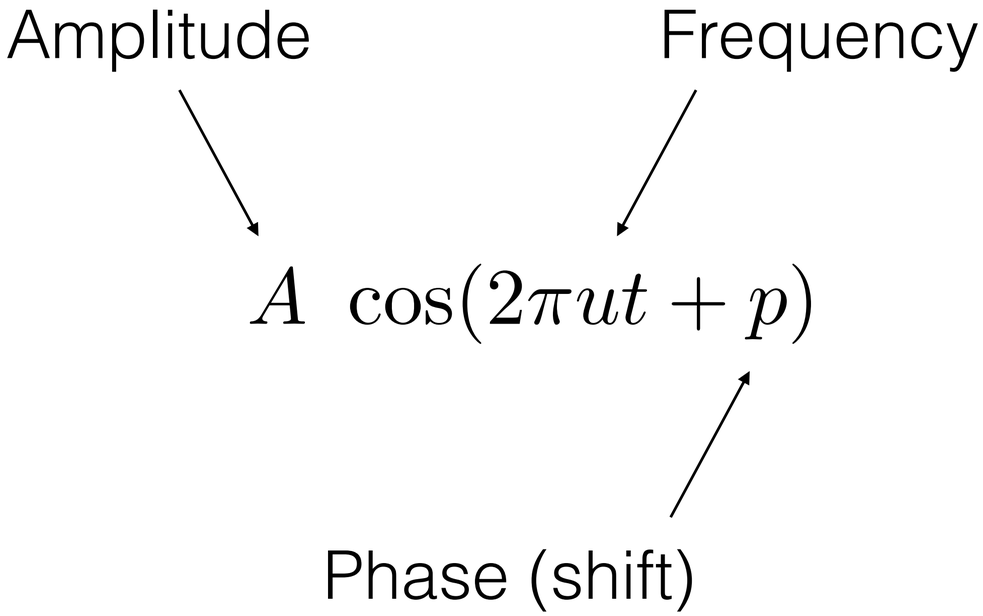
The Fourier Transform

CS 355: Introduction to Graphics and Image Processing

First, a bit about
sines and cosines…

Sinusoids

- Useful to think of sinusoids in terms of three properties:
  - their *frequency* (how often they repeat)
  - their *amplitude* (height of the peaks)
  - their *phase* (shifting left or right)
- A sine wave is just a cosine wave with phase π/2



And a bit about transforms…

Functions as Vectors

- Inner (dot) product between two vectors is the summation of the point-wise product
- Can't we do the same thing with functions?
- For continuous functions, the summation just becomes an integral
- Functions satisfy all of the mathematical requirements for "vectors"
- Can we transform *functions*?

$$\mathbf{u} \cdot \mathbf{v} = \sum_j \mathbf{u}[j] \ \mathbf{v}[j]$$

$$f(t) \cdot g(t) = \int_{-\infty}^{\infty} f(t) \ g(t) \ dt$$

Sinusoidal Basis Functions

- One set of orthonormal basis functions is the set of sines and cosines of different frequencies
- Let's use these as the basis functions for a transform

$$c(u) = \int_{-\infty}^{\infty} f(t)\ \cos(2\pi ut)\ dt$$

$$s(u) = \int_{-\infty}^{\infty} f(t)\ \sin(2\pi ut)\ dt$$

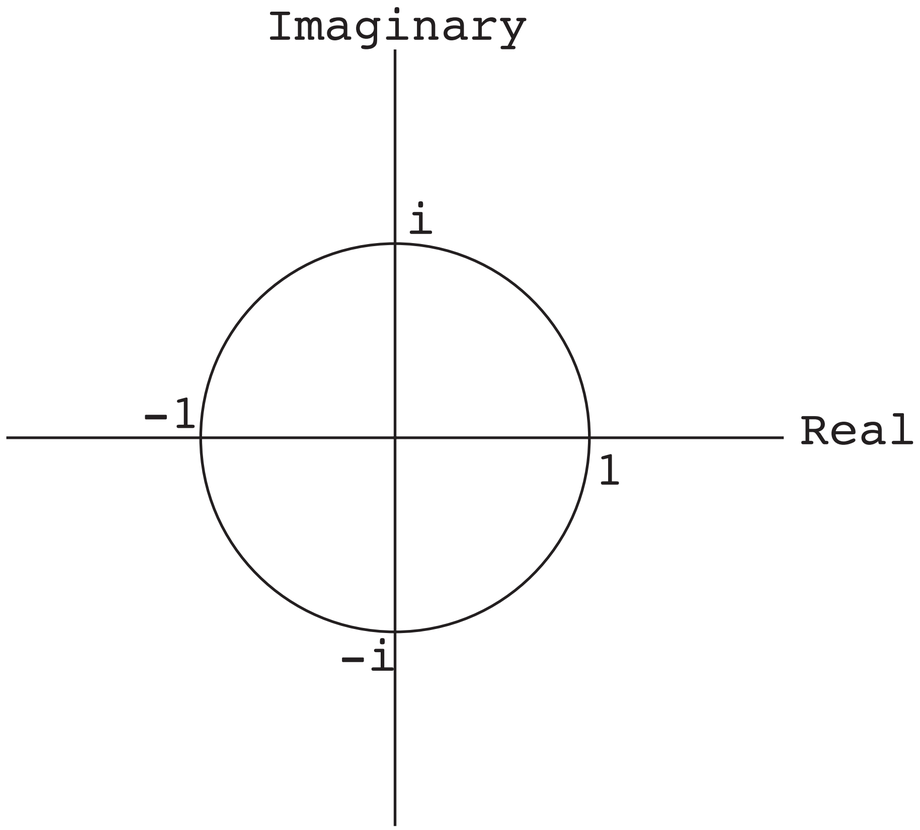
A brief detour…

Complex Numbers

- A complex number is the sum of a real number and an imaginary number:

$$a + bi$$

- Think of as a point on the *complex plane*



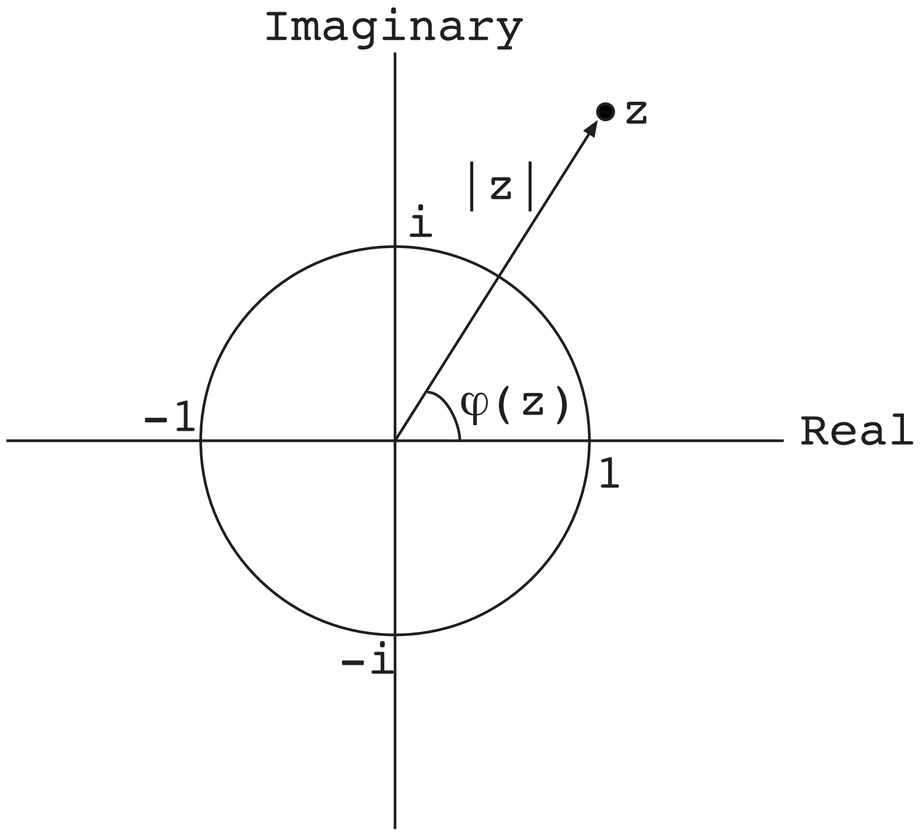
Complex Numbers

- Often useful to think about complex numbers in polar coordinates
- The magnitude is

$$\|a + bi\| = \sqrt{a^2 + b^2}$$

- And the phase (angle) is

$$\phi(a + bi) = \tan^{-1}\left(\frac{b}{a}\right)$$



So why do we care about complex numbers?

Adding Sinusoids

- When you add some amount of a cosine to another amount of a sine of the same frequency, you get a sinusoid of the same frequency

$$a\cos(2\pi ut) + b\sin(2\pi ut)$$

- If you encode the amount of cosine and the amount of sine as a complex number…

$$z = a + bi$$

  - The amplitude of the sinusoid is the magnitude of the complex number
  - The phase of the sinusoid is the phase of the complex number

$$\|z\| \ \cos(2\pi ut + \phi(z))$$

Two Ways to Make Sinusoids

- We thus have two ways to make *the same sinusoid*:
  - Mix a cosine and a sine with specific weights
  - Start with a cosine and
    - Stretch it by the square root of the sum of the squares of the sine and cosine weights
    - Shift it by the arctangent of the ratio of the sine weight to the cosine weight

Sinusoidal Weights as Complex Numbers

- Useful to encode the coefficients from projecting onto sinusoidal basis functions as a single complex number

$$c(u) = \int_{-\infty}^{\infty} f(t)\ \cos(2\pi u t)\ dt$$

$$s(u) = \int_{-\infty}^{\infty} f(t)\ \sin(2\pi u t)\ dt$$

$$F(u) = c(u) - i\ s(u)$$

* Note the minus sign here when combining the two, this comes from doing linear algebra with complex quantities

The Fourier Transform

- An encode as a function f(t) as a weighted sum of sines and cosines where the weights are given by a complex-valued function F(u)
- Can think of as an operator
- Written as

$$F(u) = \mathcal{F}(f(t))$$

$$c(u) = \int_{-\infty}^{\infty} f(t)\ \cos(2\pi u t)\ dt$$

$$s(u) = \int_{-\infty}^{\infty} f(t)\ \sin(2\pi u t)\ dt$$

$$F(u) = c(u) - i\ s(u)$$

The Inverse Fourier Transform

- Can invert the transformation to get f(t) from F(u) by simply adding the sines and cosines back up with the respective weights
- Can also think of as an operator
- Written as

$$f(t) = \mathcal{F}^{-1}(F(u))$$

$$F(u) = a(u) + i\ b(u)$$

$$f(t) = \int_{-\infty}^{\infty} a(u)\ \cos(2\pi ut)\ du + \int_{-\infty}^{\infty} b(u)\ \sin(2\pi ut)\ du$$

The Discrete Fourier Transform

- What about a finite-length, sampled signal?
- The Fourier Transform assumes an infinite-length signal
  (impossible to work with on computers)
- What should we assume about the signal before and after we record it?
- Since we're decomposing it into a sum of periodic functions,
  let's assume it's one period of an infinite periodic function

Period Functions

- For a periodic signal with period N units, all of the underlying frequencies must also repeat over the period N
- So, each component frequency must be a multiple of the frequency of the periodic signal itself:
- There are no more than N components for a signal with period N samples!

$$\frac{0}{N}, \frac{1}{N}, \frac{2}{N}, \frac{3}{N} \cdots \frac{N-1}{N}$$

The Discrete Fourier Transform

- Same idea as the Fourier Transform
- For a finite with N samples:
  - N discrete frequencies *u/N*
  - Sum over the N discrete samples
- What would the code look like?
- What is the complexity?

$$c[u] = \frac{1}{N} \sum_{u=0}^{N-1} f[t] \ \cos(2\pi ut/N)$$

$$s[u] = \frac{1}{N} \sum_{u=0}^{N-1} f[t] \ \sin(2\pi ut/N)$$

$$F[u] = c[u] - i \ s[u]$$

The Fast Fourier Transform

- The Fast Fourier Transform does exactly the equivalent mathematically
- Divide-and-conquer algorithm provides greater efficiency

DFT $O(N^2)$

FFT $O(N \log N)$

Using the FFT

- Pass in an array of length N
- Returns back an array of length N of type complex
  - The real part of each number is how much of a cosine of that frequency there is
  - The imaginary part of each number is how much of a sine of that frequency there is

Interpreting the Complex Numbers

- Can think of in Cartesian form:
  - The real part of each number is how much of a cosine of that frequency
  - The imaginary part of each number is how much of a sine of that frequency
- Or in polar form:
  - The magnitude of each number is how much of that frequency there signal
  - The phase of each number is the relative shift (from a cosine) of each sinusoid

Negative Frequencies

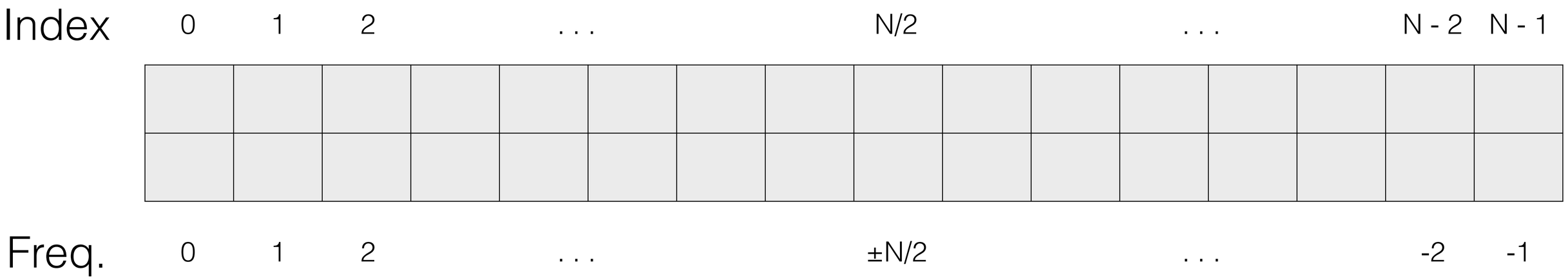
- Because of their repetitive nature
  - cosines are symmetric
  - sines are antisymmetric
  - discrete ones repeat modulo N

$$\cos(-2\pi u/N) = \cos(2\pi u/N)$$

$$\sin(-2\pi u/N) = -\sin(2\pi u/N)$$

$$N - u \equiv -u \mod N$$

Storage of the DFT Results



Because of symmetry, the last half of the array is a mirrored (negated) copy of the first half

$$\cos(-2\pi u/N) = \cos(2\pi u/N)$$

$$\sin(-2\pi u/N) = -\sin(2\pi u/N)$$

Implications

- Because of symmetry
  - Last half of the real part is a mirrored copy of the first half
  - Last half of the imaginary part is a negated mirrored copy
- There are really only N basis functions, as there should be, not 2N
- Only really computing frequencies up to N/2
  - Twice the number of samples are there are frequencies
  - “Sample at twice the highest frequency in the signal…”
  - This is the same as Shannon’s sampling theorem!

Useful Python Functions

- `np.fft.fft` - forward FFT (complex array in, complex array out)
- `np.fft.ifft` - inverse FFT (complex array in, complex array out)
- `np.absolute` - returns the magnitude of a complex number (or the absolute value of a real one)
- Normal mathematical operators (+, -, *, /) work on complex numbers as well as integer, floating point ones
- Tip: a real number is a complex one with imaginary part equal to zero

Why Are We Doing This Again?

- The Fourier Transform of a signal lets you analyze the mix of frequencies in it
- Can manipulate the transformed signal and then transform it back! (that's the topic for the next class)

Coming up...

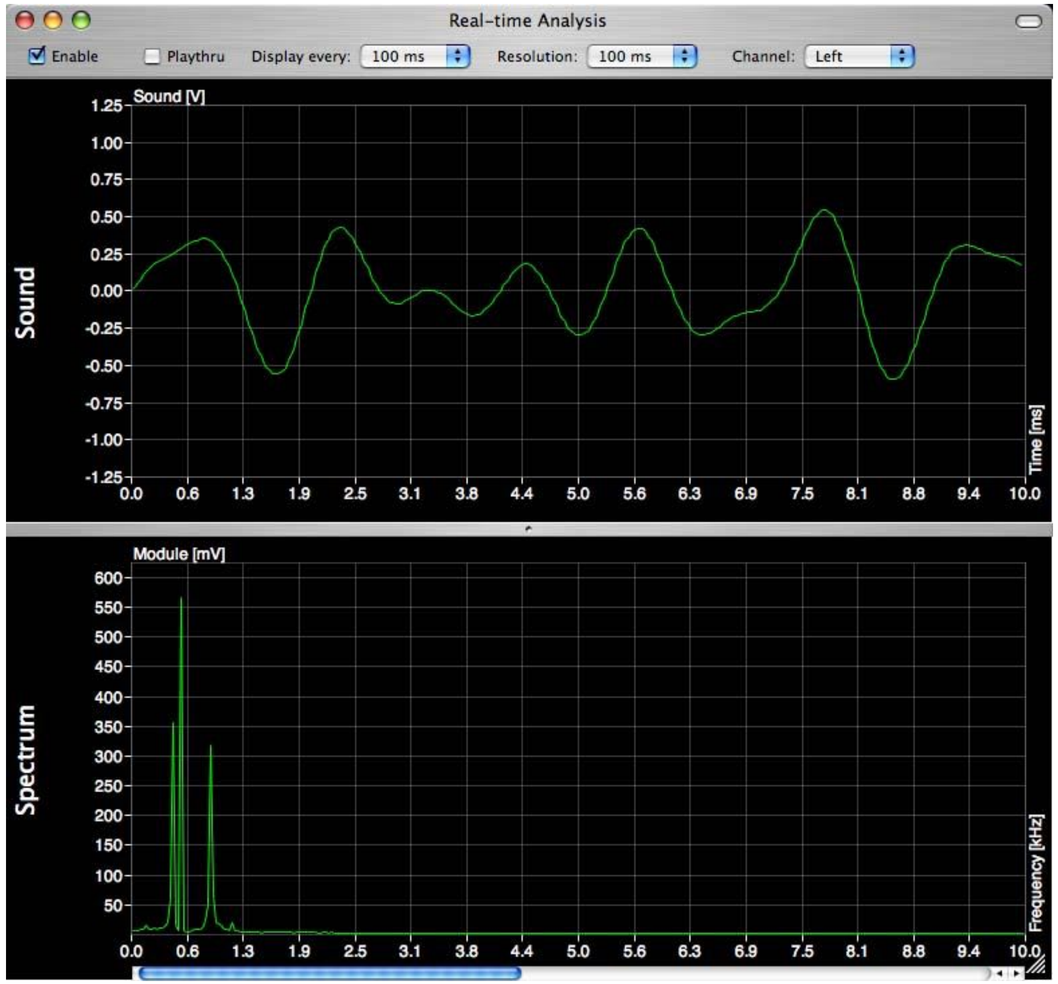
- Examples and properties
- Filtering in the frequency domain
- The Convolution Theorem (wait, what? convolution?)
- 2-D FFT and image filtering in the frequency domain

Frequency-Domain Processing

CS 355: Introduction to Graphics and Image Processing

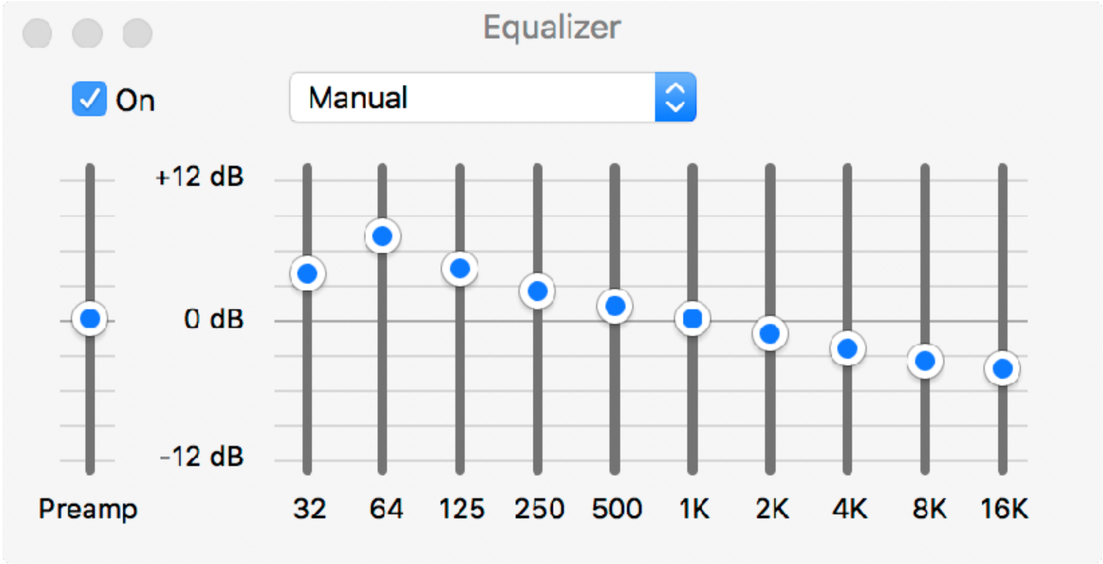
Time Domain vs. Frequency Domain

- For sampled functions of time, we talk about manipulating these samples in the **time domain**
- Earlier in the semester, we learned ways to manipulate images in the **spatial domain**
- Let's talk about manipulating them in the **frequency domain**

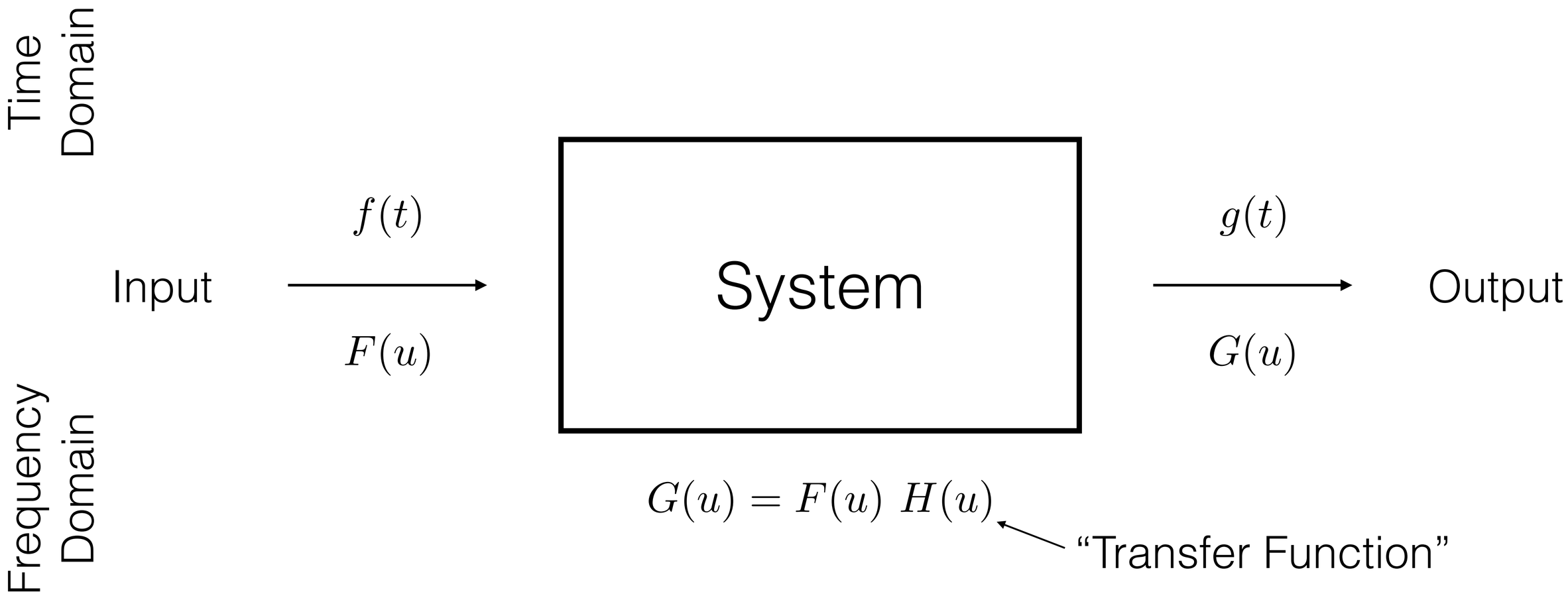


Frequency-Domain Processing

- The simplest way to process signals in the frequency domain is to
  - Selectively amplify (boost) certain frequencies
  - Selectively attenuate (weaken) other frequencies



Common Notation



The Transfer Function

- Intuition: how different frequencies "transfer" from input to output
- Remember:
  - Input F(u) is complex
  - Output G(u) is also complex
- If transfer function H(u) is **real-valued**, it multiplies both the real (cosine) part and the imaginary (sine) part equally
  - Magnitude/amplitude is scaled (amplification/attenuation)
  - Phase is left unchanged

$$G(u) = F(u)\ H(u)$$

The Transfer Function

- If H(u) is **complex-valued**, it multiplies the real (cosine) part and the imaginary (sine) part differently
  - The magnitudes multiply (amplification/attenuation)
  - The phases add (shifts different frequencies potentially differently)

$$G(u) = F(u)\ H(u)$$

The Transfer Function

$$G(u) = F(u)\ H(u)$$

Modulation Transfer Function

$$|H(u)|$$

Selectively amplifies/attenuates

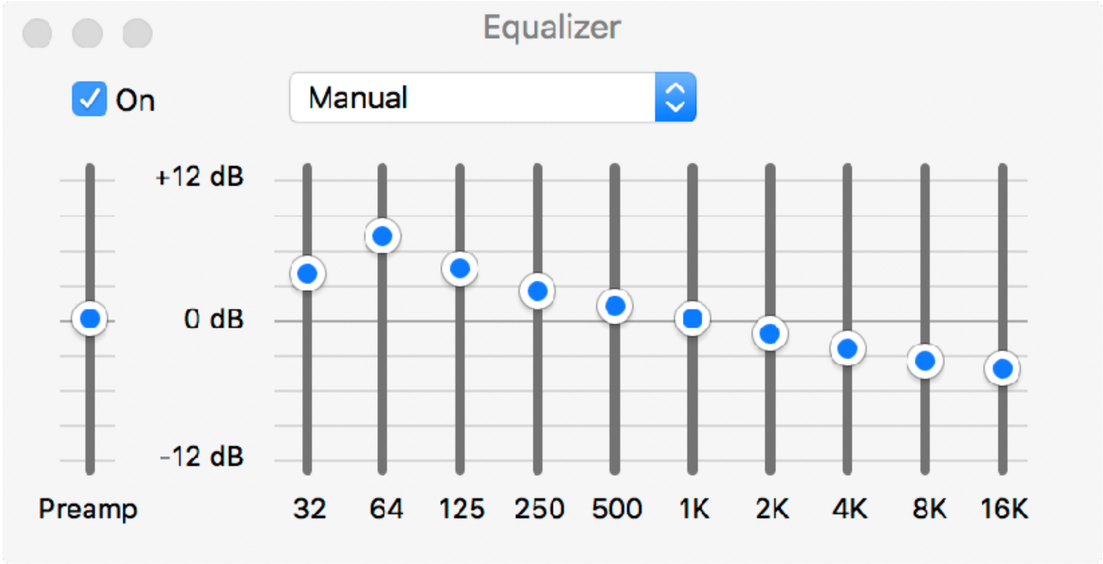
Phase Transfer Function

$$\phi(H(u))$$

Selectively shifts different frequencies

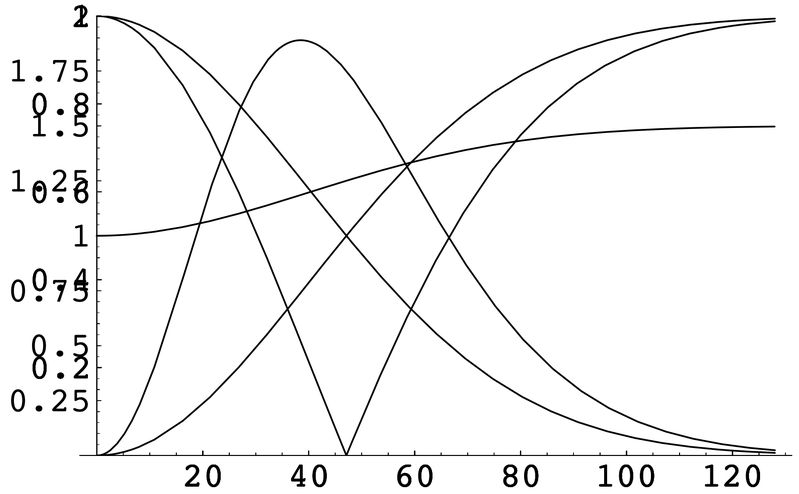
Filtering

- Designing and applying a transfer function to a signal in the frequency domain it is called **filtering**
- You can design H(u) to be whatever you want it to be!



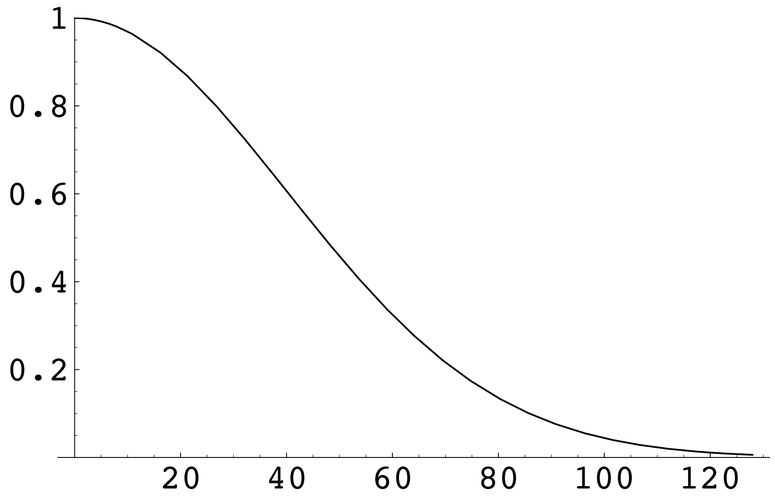
Types of Filters

Low-Pass	Keeps low frequencies, attenuates higher ones
High-Pass	Keeps high frequencies attenuates lower ones
High-Boost	Keeps lower frequencies, boosts higher frequencies
Band-Pass	Attenuates frequencies above or below a target band
Band-Suppress	Attenuates frequencies in a target band



Low-Pass Filters

- Lets **low** frequencies **pass** through
- Attenuates higher frequencies
- Examples:
  - Sound through air
  - Sound through other materials
  - RF signals through air
  - Electronic signals through wire

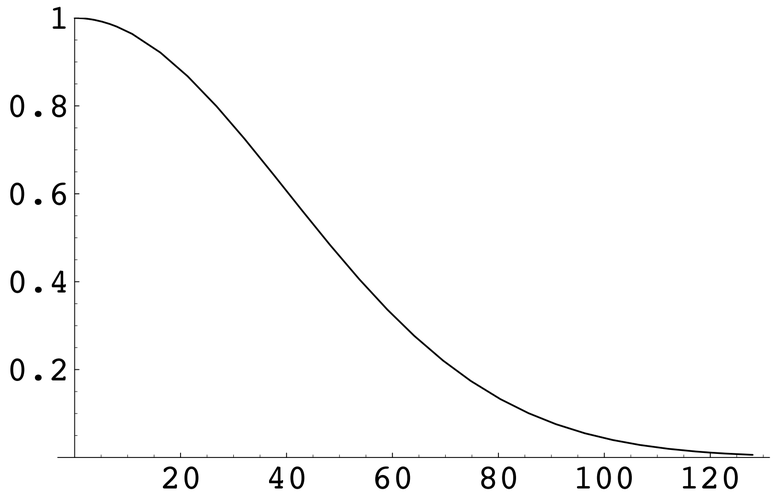


Example: Gaussian Low-Pass Filter

- One common form of a low-pass filter is the Gaussian

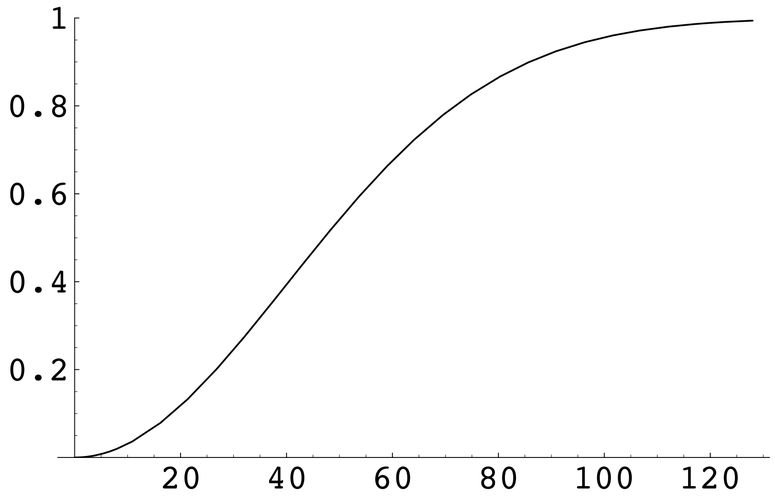
$$H(u) = e^{-\frac{1}{2}u^2/{u_c}^2}$$

Cutoff frequency



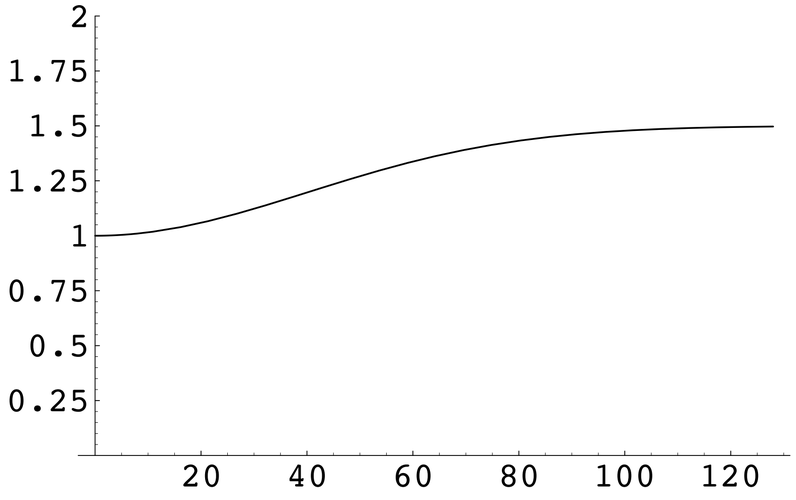
High-Pass Filters

- Lets **high** frequencies **pass** through
- Attenuates lower frequencies
- Not very common in nature but useful in electronics or signal processing
- Often 1.0 minus a low-pass filter



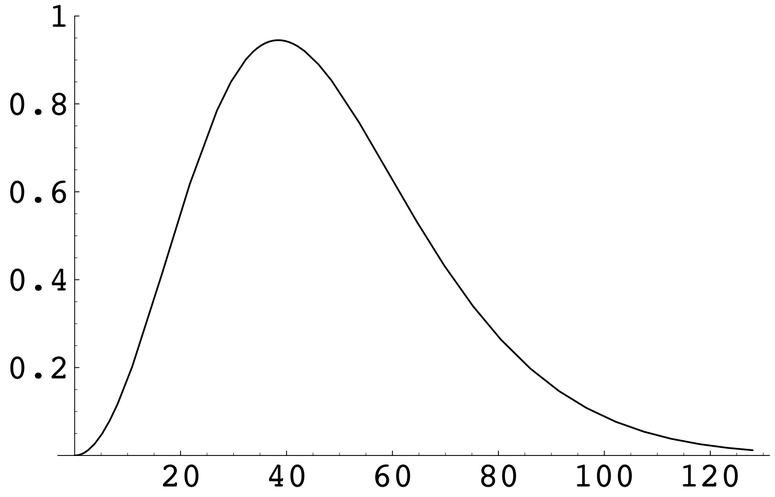
High-Boost Filters

- Boosts high frequencies
- Useful for “undo-ing” low-pass filters
- Can boost any other part of the frequency domain as well
- Often 1.0 plus a high-pass filter



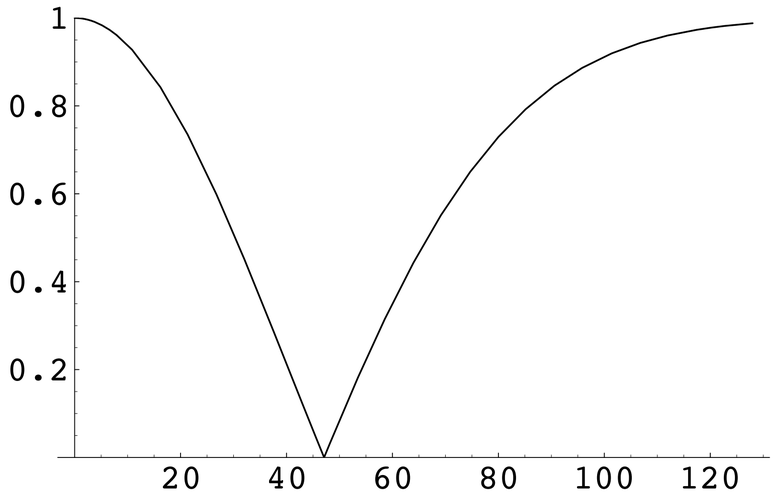
Band-Pass Filters

- Lets frequencies in some target **band pass** through
- Attenuates higher/lower frequencies
- Eg., difference of two low-pass filters



Band-Suppresses Filters

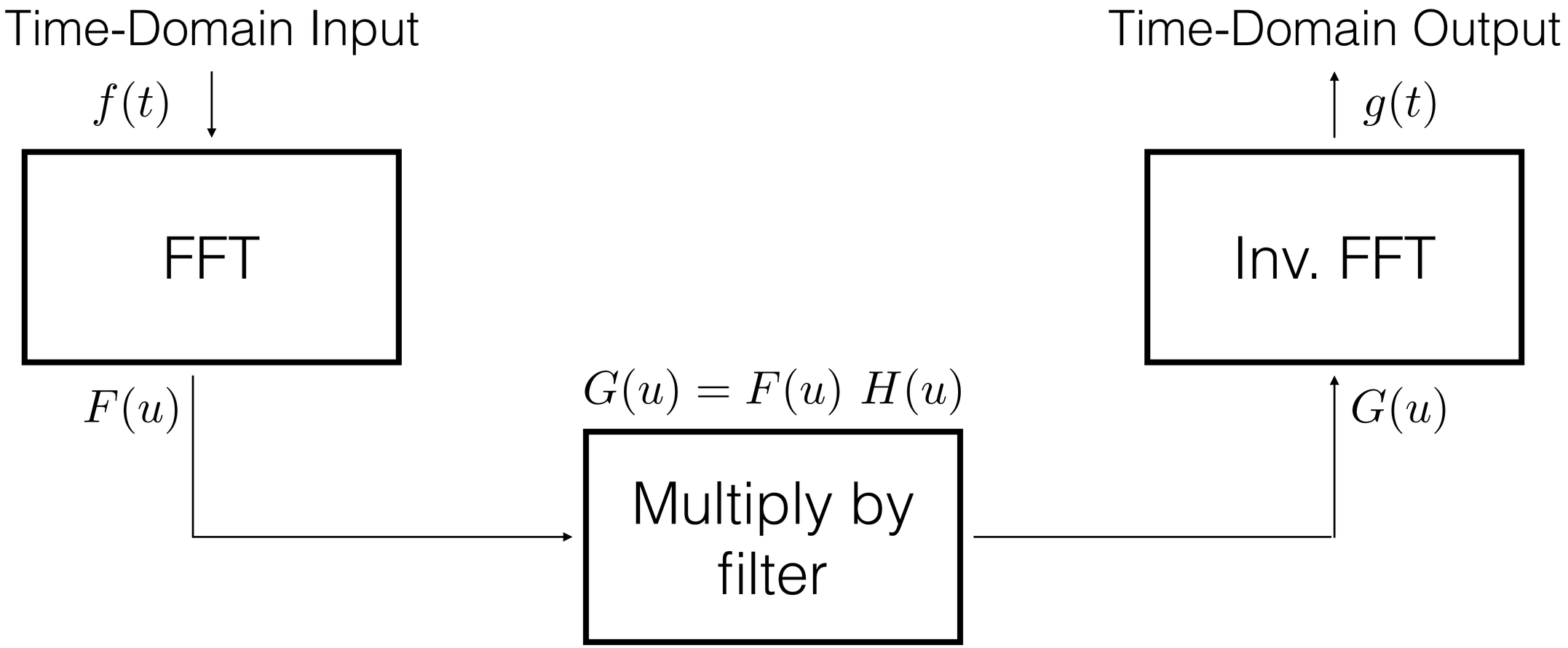
- **Suppresses** frequencies in a target **band**
- Preserves higher/lower frequencies



Phase Filtering

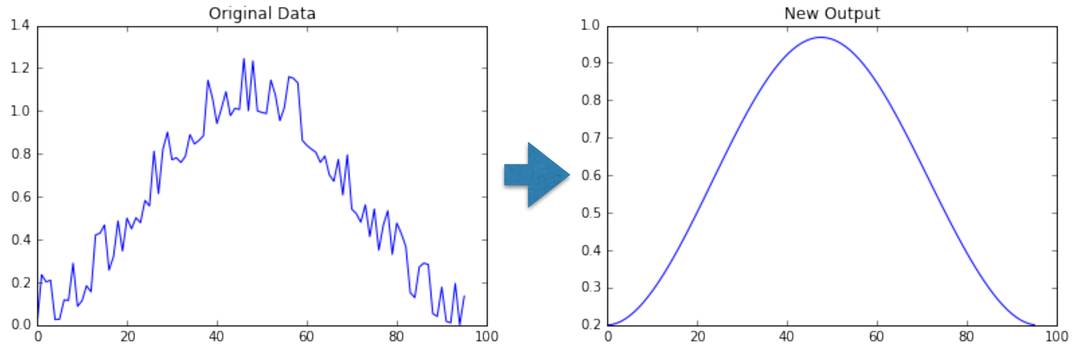
- Time delay
- Example: transmission of electronic signals through wires
  - Different frequencies pass at different speeds through the same media
  - Can become noticeable over long distances
  - Solution: periodically gather, re-sync the phases, retransmit

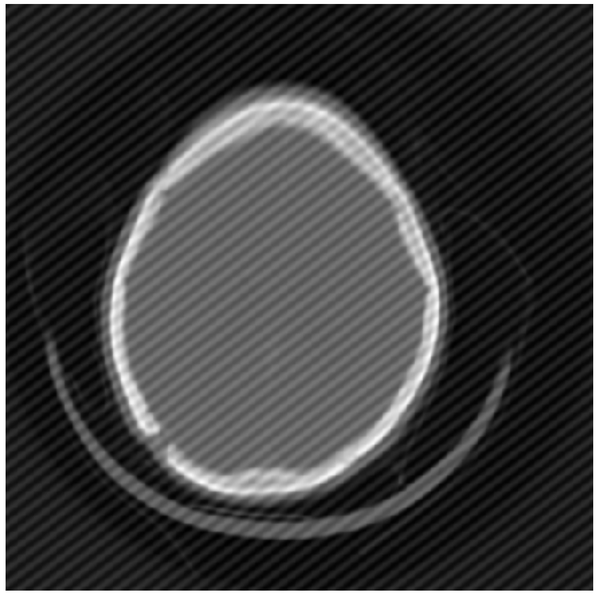
Implementing Filtering Digitally



Lab 10

- Filtering 1-D signals
  - Synthetic example (noisy function)
  - Audio example
- Filtering 2-D images
  - Smoothing/sharpening
  - Isolating and removing a single-frequency interference pattern





Coming up...

- 2-D FFT and image filtering in the frequency domain
- And then we’re done!

2D Fourier Transform

CS 355: Introduction to Graphics and Image Processing

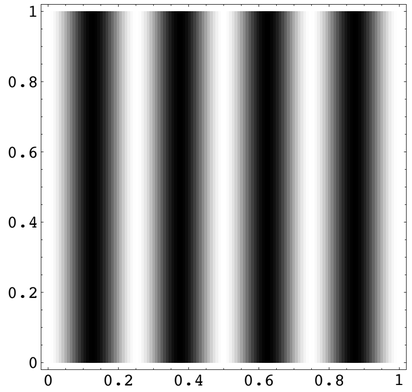
The 2D Discrete Fourier Transform

- Uses basis functions of two variables
  - u = frequency in the x direction
  - v = frequency in the y direction

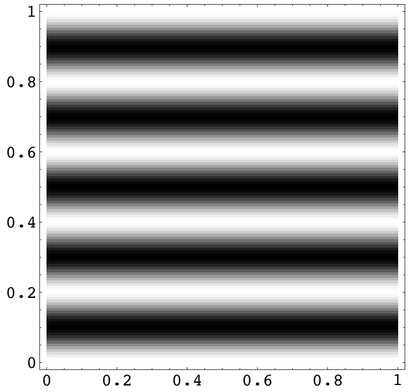
For an N x M image:

$$\cos(2\pi(ux/M + vy/N))$$

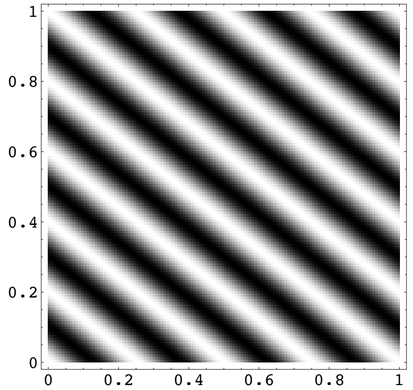
$$\sin(2\pi(ux/M + vy/N))$$



u = 4, v = 0



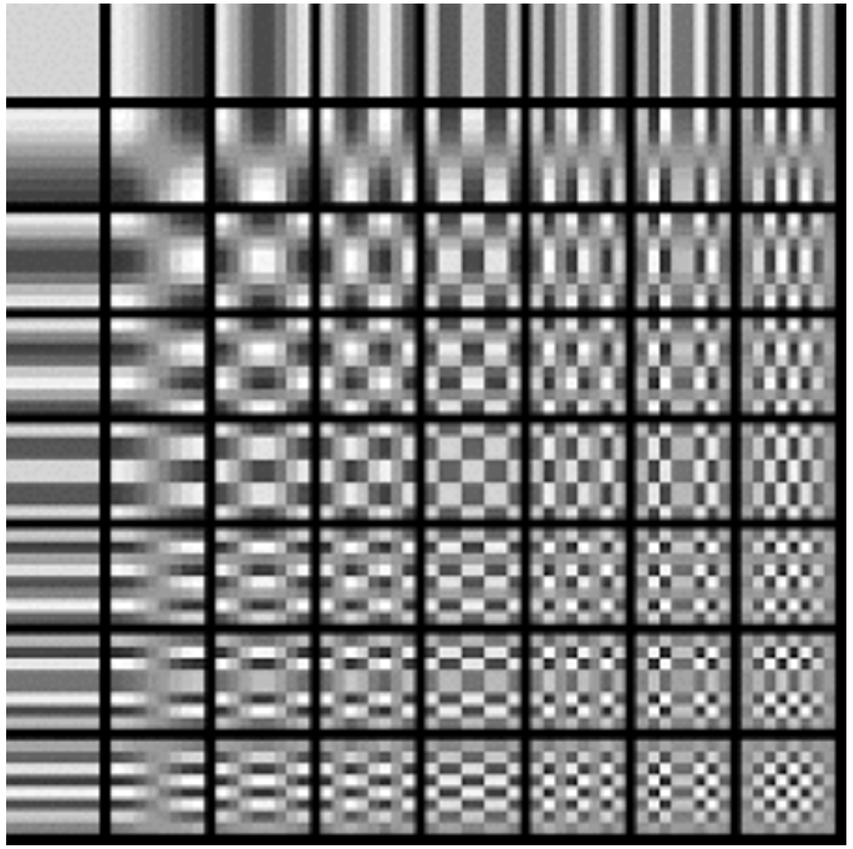
u = 0, v = 5



u = 4, v = 5

The 2D Discrete Fourier Transform

- Example: 64 basis images can be combined in a weighted sum to form **any** 8 x 8 image



* Technically, these are for the DCT, a close cousin to the DFT

Spatial Frequencies

- We call these different image components “spatial frequencies”
- Analogous to 1-D frequencies like audio
- Intuition:
  - low frequencies are slow, gradual changes
  - high frequencies are rapid changes (e.g., textures, edges)

The 2D Discrete Fourier Transform

$$
\begin{aligned}
F[u,v] &= \frac{1}{NM} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f[x,y] \ \cos(2\pi(ux/M + vy/N)) \ dx \ dy \\
&+ i \ \frac{1}{NM} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f[x,y] \ \sin(2\pi(ux/M + vy/N)) \ dx \ dy
\end{aligned}
$$

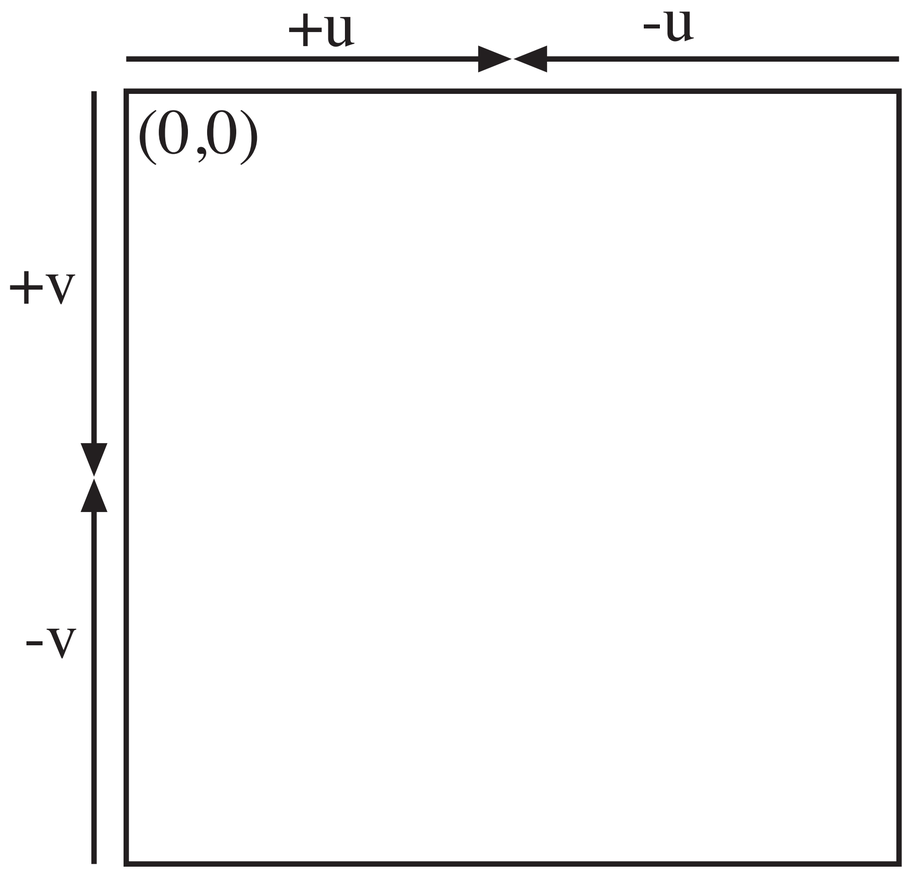
What's does the code look like?

What's the computational complexity?

The 2D Fast Fourier Transform

	DFT	FFT
1-D	$O(N^2)$	$O(N \log N)$
2-D	$O(N^4)$	$O(N^2 \log N)$

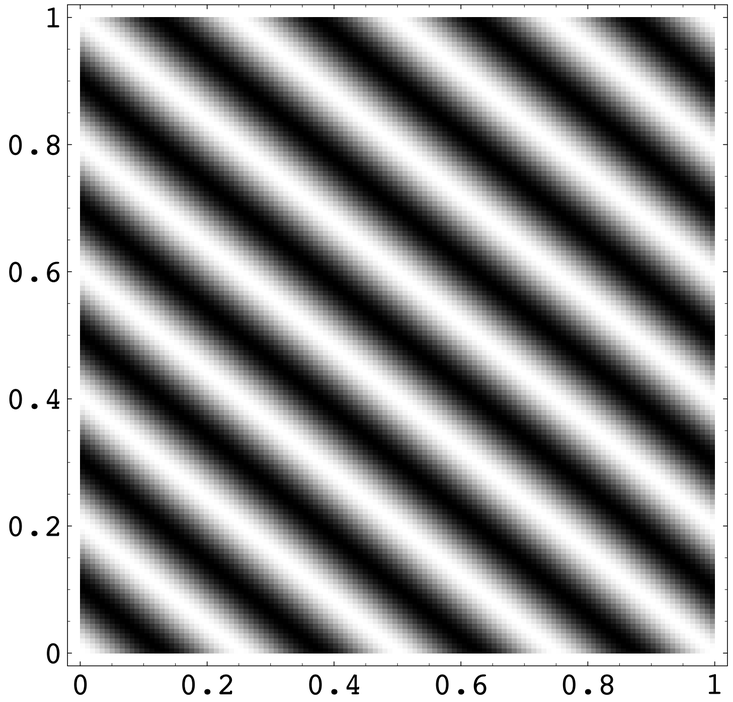
2D DFT Storage



(like 1-D, it's often useful to shift it to better visualize)

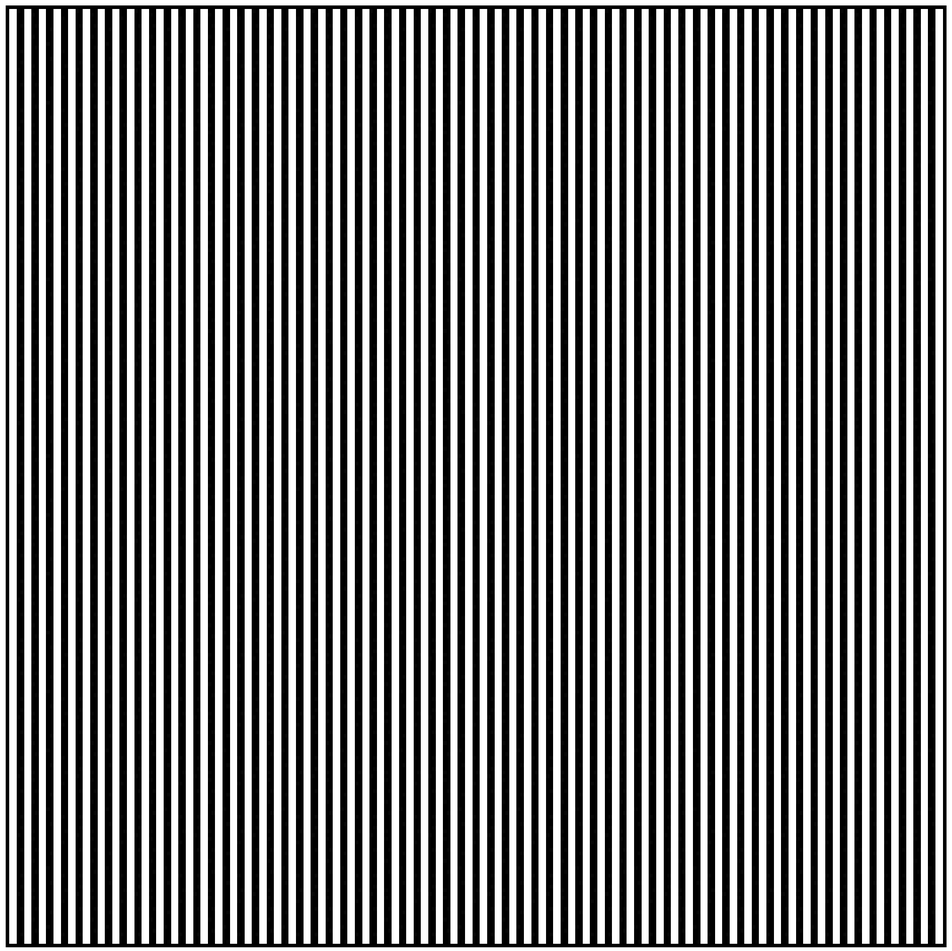
Interpreting the 2D DFT

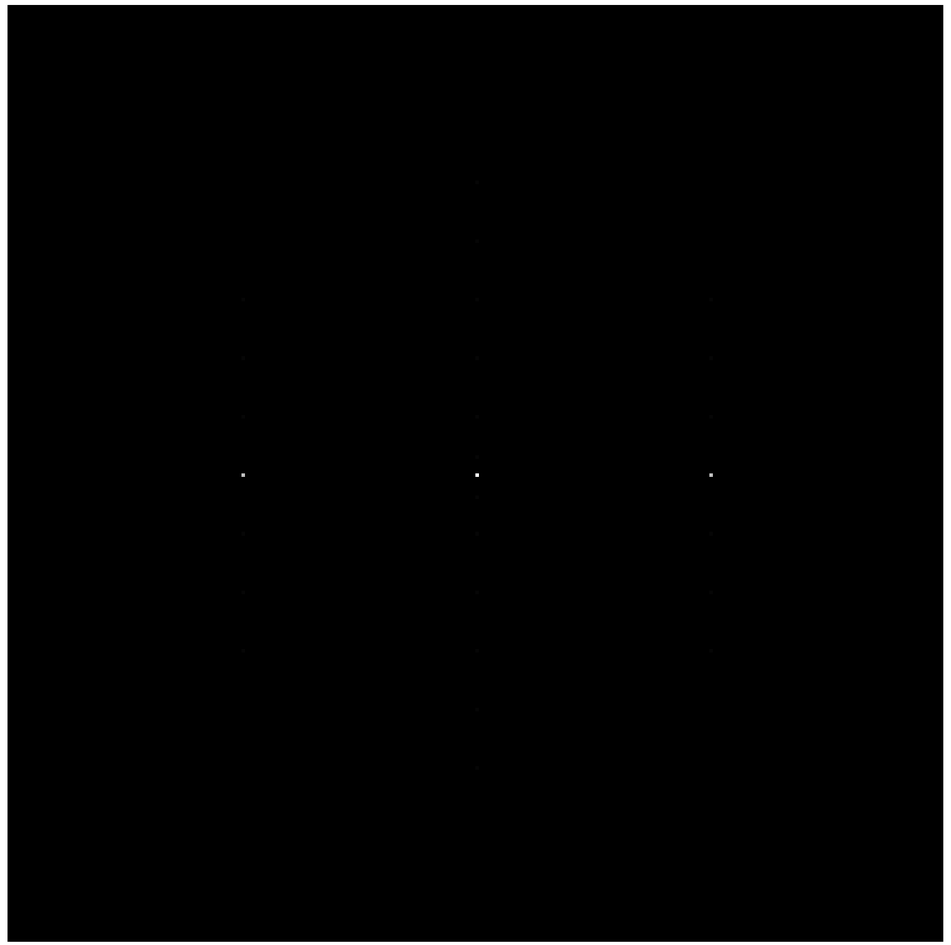
- Frequencies in x and y directions
  - u = frequency in the x direction
  - v = frequency in the y direction
- A sinusoidal pattern in a single direction
- Useful property: rotating the image rotates its Fourier Transform



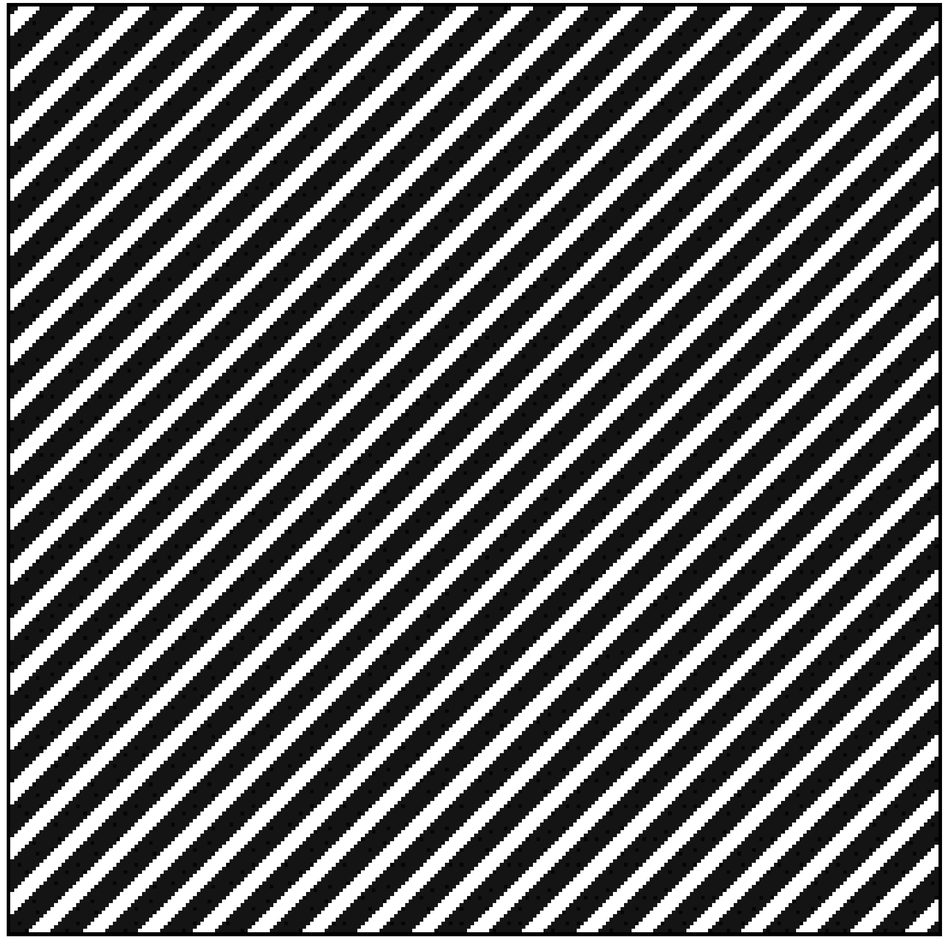
u = 4, v = 5

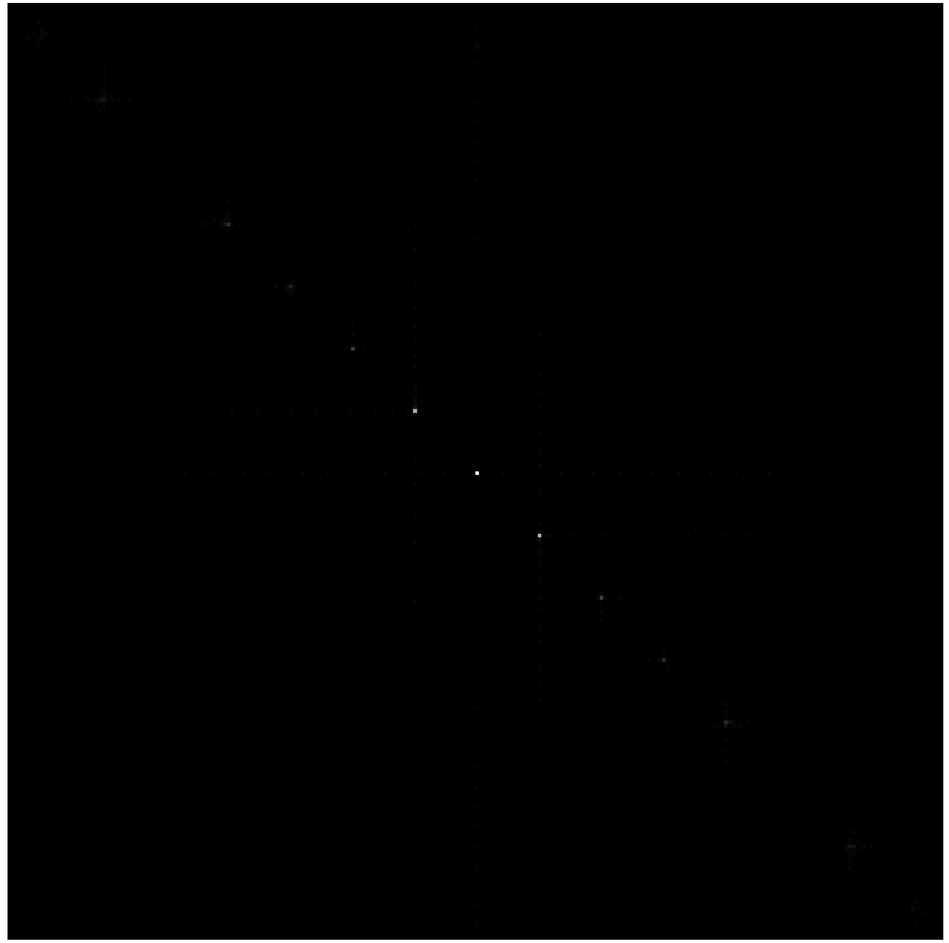
Examples





Examples

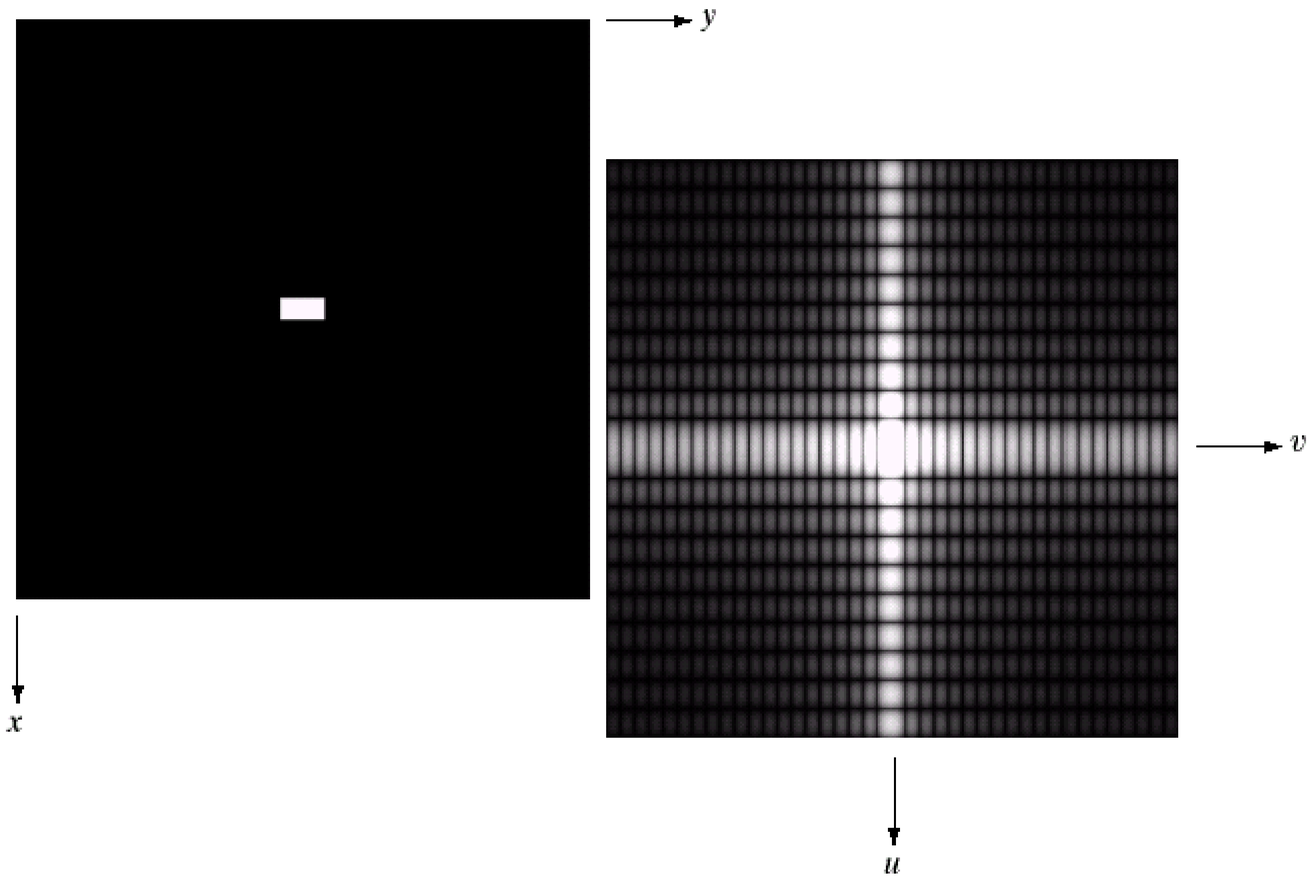




Examples

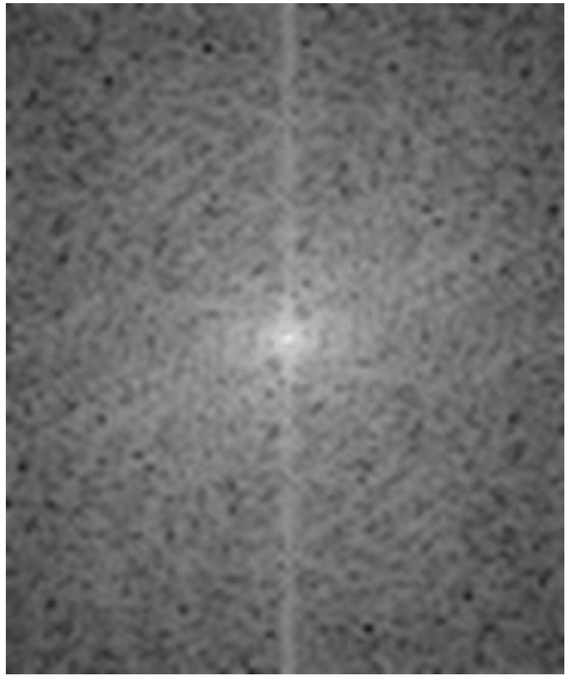
a b

FIGURE 4.3
(a) Image of a 20×40 white rectangle on a black background of size 512×512 pixels.
(b) Centered Fourier spectrum shown after application of the log transformation given in Eq. (3.2-2). Compare with Fig. 4.2.



Another Example

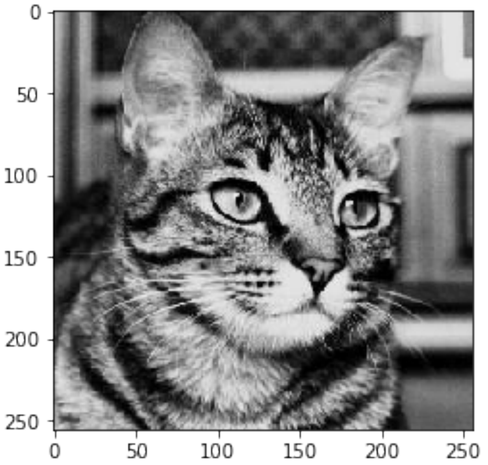
Jean Baptiste
Joseph Fourier

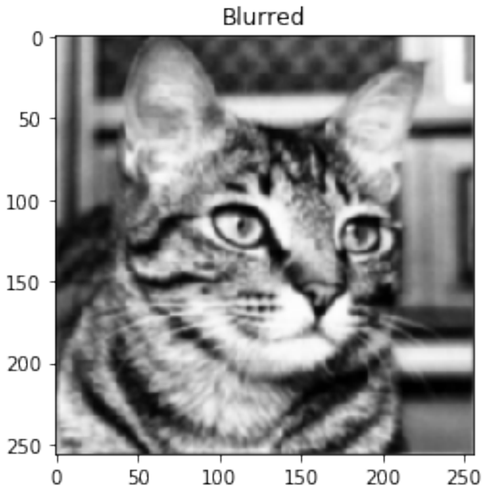


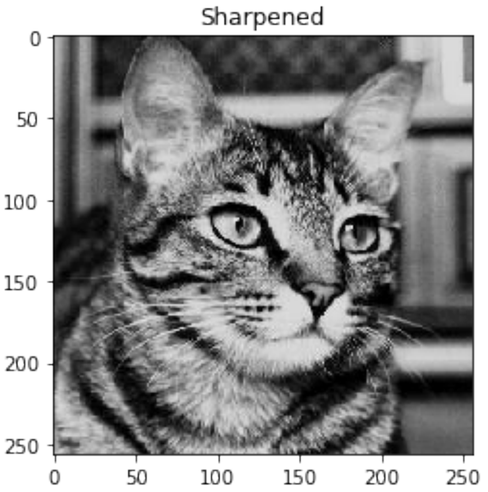
His Fourier Transform

Filtering Images in the Frequency Domain

- Just like we can filter 1-D signals, we can also filter 2-D images
  - Fourier transform the image
  - Multiply by the filter
  - Inverse Fourier Transform
- Low-pass filtering = blurring
- High-pass filtering = edge detection
- High-boost filtering = sharpening





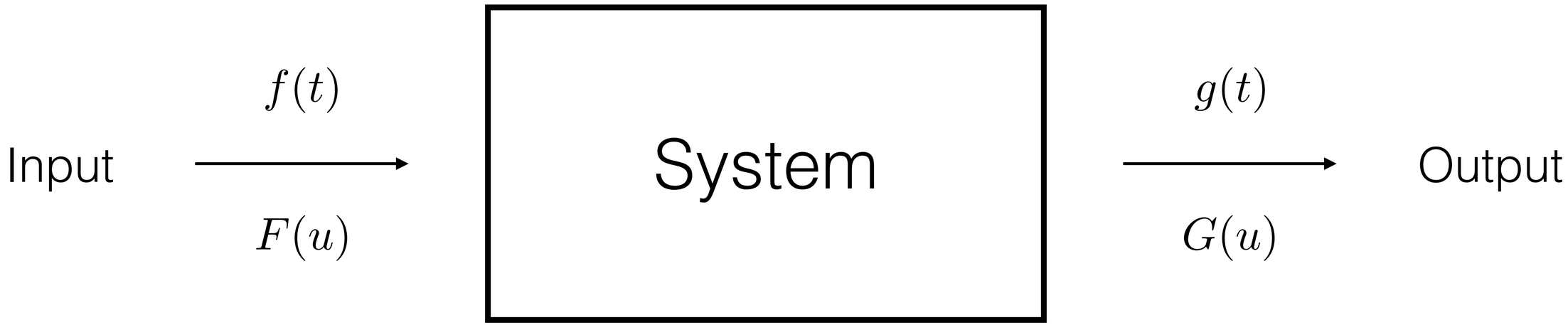


A really important detour…

Time Domain vs. Frequency Domain

Time Domain

What's the equivalent?



Frequency Domain

$$G(u) = F(u)\ H(u)$$

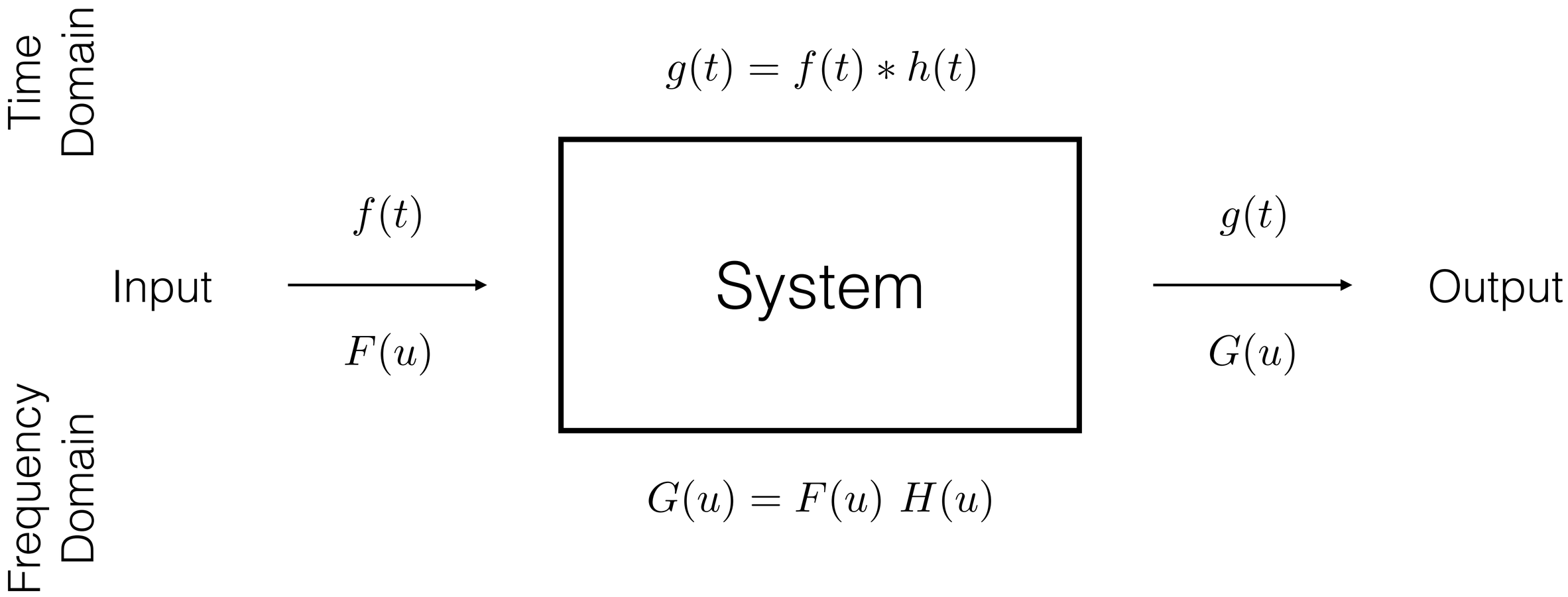
The Convolution Theorem

- **Convolution in one domain** is **multiplication in the other** and vice versa
  - Convolution in the time domain and multiplication in the frequency domain are equivalent
  - Multiplying two signals in the time domain convolves their FTs (useful for lots of things)
- Relationship between convolution kernels and transfer functions?

$$f(t) * g(t) = \mathcal{F}(f(t)\ \mathcal{F}(g(t))$$

$$f(t)\ g(t) = \mathcal{F}(f(t) * \mathcal{F}(g(t))$$

Time Domain vs. Frequency Domain

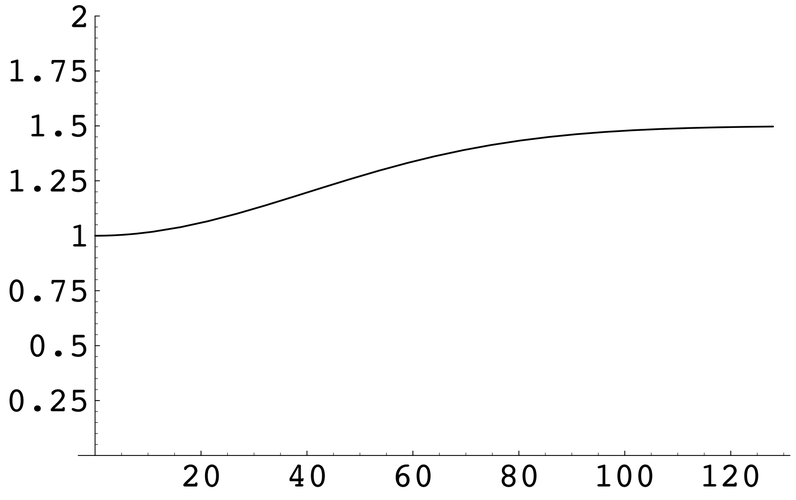


Example: High Boost Filters

- Can construct a high-boost filter by
  - Low-pass filtering the signal
  - Subtracting it from the original
  - Adding a fraction of that back to the original

$$H(u) = 1 + \alpha(1 - e^{-\frac{1}{2}u^2/{u_c}^2})$$

- Sound familiar?



The Need For Speed…

- Convolution of an N x N image with a K x K kernel:

$$O(N^2 K^2)$$

- FFT the signal, FFT the kernel, inverse FFT:

$$O(N^2 \log N)$$

- For large kernels, doing it in the frequency domain can be faster!

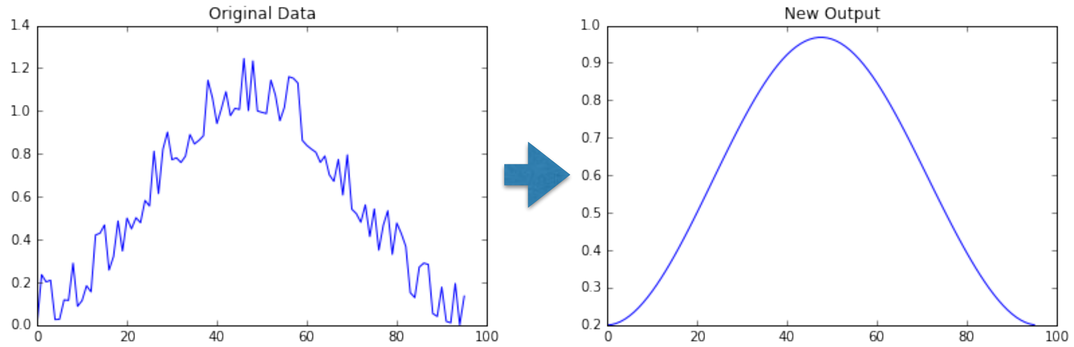
- General rule of thumb, the break-even point is 7 x 7 kernels

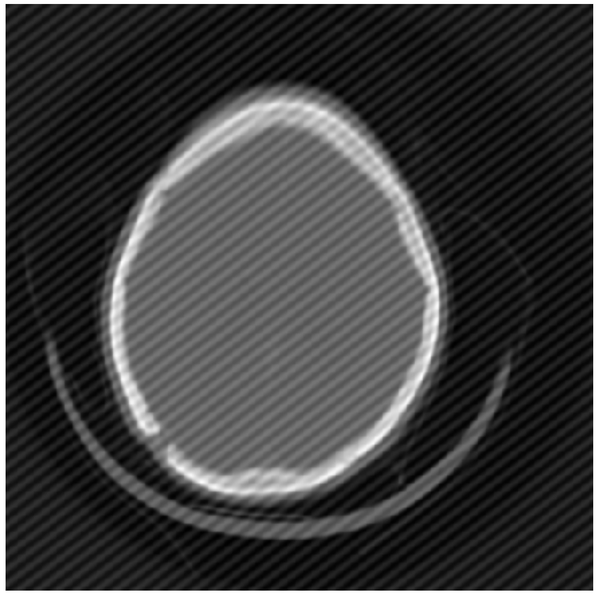
Design vs. Implementation

- We can design in either domain
  - Spatial filters in (masks or kernels) in the time/spatial domain
  - Frequency filters in the frequency domain
- And we can implement them in either domain
  - Convolution in the time/spatial domain
  - Multiplication in the frequency domain

Lab 10

- Filtering 1-D signals
  - Synthetic example (noisy function)
  - Audio example
- Filtering 2-D images
  - Smoothing/sharpening
  - Isolating and removing a single-frequency interference pattern





Coming up...

- Other uses of these ideas in Computer Science
- Applications in Computer Graphics (and CS 455)
- Applications in Computer Vision (and CS 450)

Other Applications in CS

CS 355: Introduction to Graphics and Image Processing

Math Applications in CS

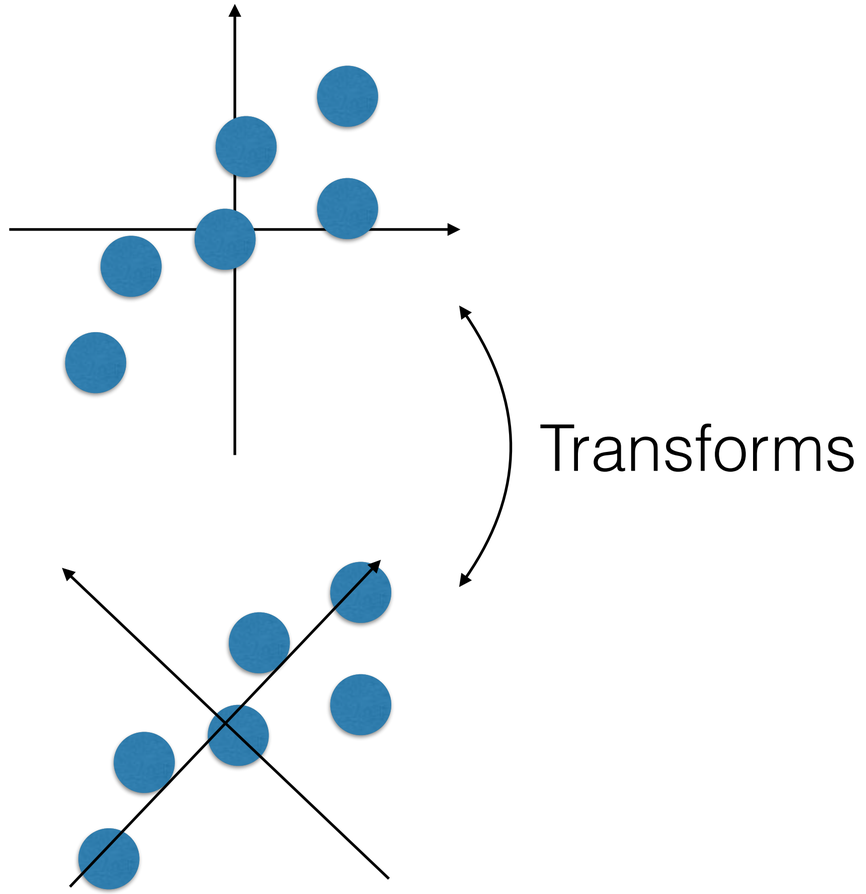
- All the “discrete math” from CS 235, 252, etc.
  - Sets, relations, functions, …
- Linear algebra
  - Geometric transformations
  - Data transforms
  - Systems of equations
  - Eigensystems

Data as Vectors

Lots of things can be thought of as points/vectors in some space

Data Transforms

- Common pattern in data analysis:
  - Represent data as vectors
  - Convert to a different coordinate system
  - Analyze (or change!) while in that coordinate system
  - Convert back (if needed)



Data Transforms

- Same form as any other change of coordinates
  - Transform using dot products
  - Convert back using weighted sum

$$g[i] = \sum_k f[k]\ e_i[k]$$

$$f[k] = \sum_i g[i]\ e_i[k]$$

Fourier Analysis

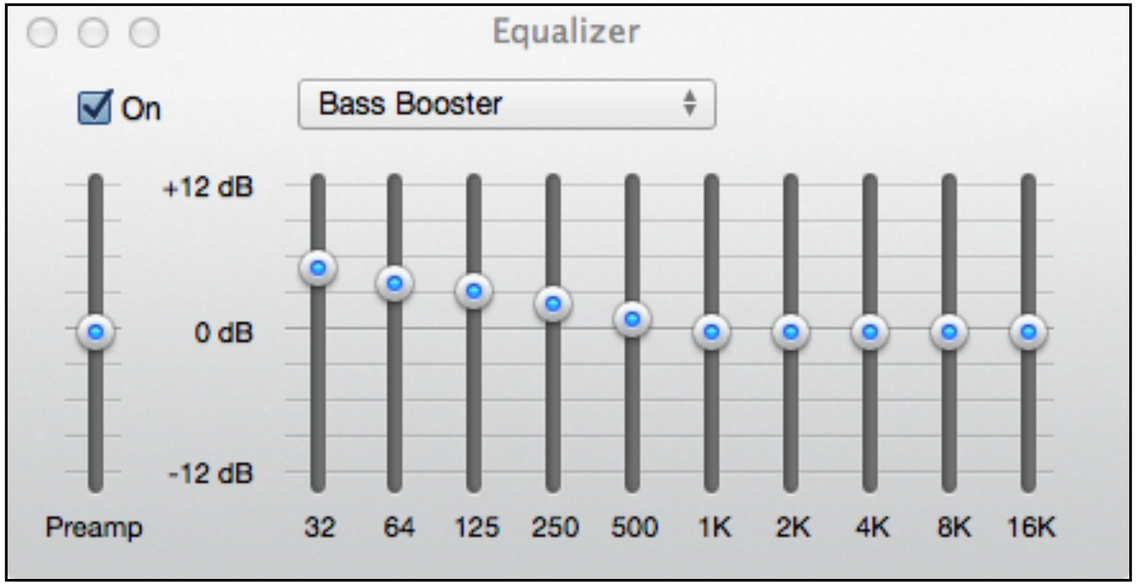
- Sampled sines and cosines of different frequencies form an orthonormal basis set
- Can decompose any waveform into a weighted sum of sines and cosines of different frequencies
- Great for analysis, manipulation, etc.

$$c[u] = \frac{1}{N} \sum_{u=0}^{N-1} f[t] \ \cos(2\pi ut/N)$$

$$s[u] = \frac{1}{N} \sum_{u=0}^{N-1} f[t] \ \sin(2\pi ut/N)$$

Frequency Manipulation

- Can use frequency-based representation to do manipulation
  - Boost bass/treble
  - Boost typical range of human speaking (hearing aides do this)
  - Suppress unwanted sounds
  - Smoothing / sharpening
  - And lots more…



Audio Compression

- Fact: your ear doesn't hear all frequencies equally well
  (and it's different for everybody)
- Idea: don't not spend as many bits of precision on the ones we don't as hear well anyway

Audio Compression

- Compression:
  - Convert to a frequency-based representation
  - Use more bits to store the coefficients for the frequencies we hear better; fewer for the ones we don’t hear well
  - Store in this form
- Decompression:
  - Use lossy coefficients and convert back to a PCM representation
  - Play!

This is how MP3 compression works!

Image Compression

- Can we do something similar with images?
- Fact: your eye is less accurate in sensing very rapid changes in brightness across an image (fine texture)
- Idea: use the same approach in the 2D frequency domain for images
- This is the basis of JPEG
  (uses Discrete Cosine Transform instead of Fourier)

Classification

- Simple problem:
  - Two classes of things, with lots of examples of each
  - New thing — what kind is it?

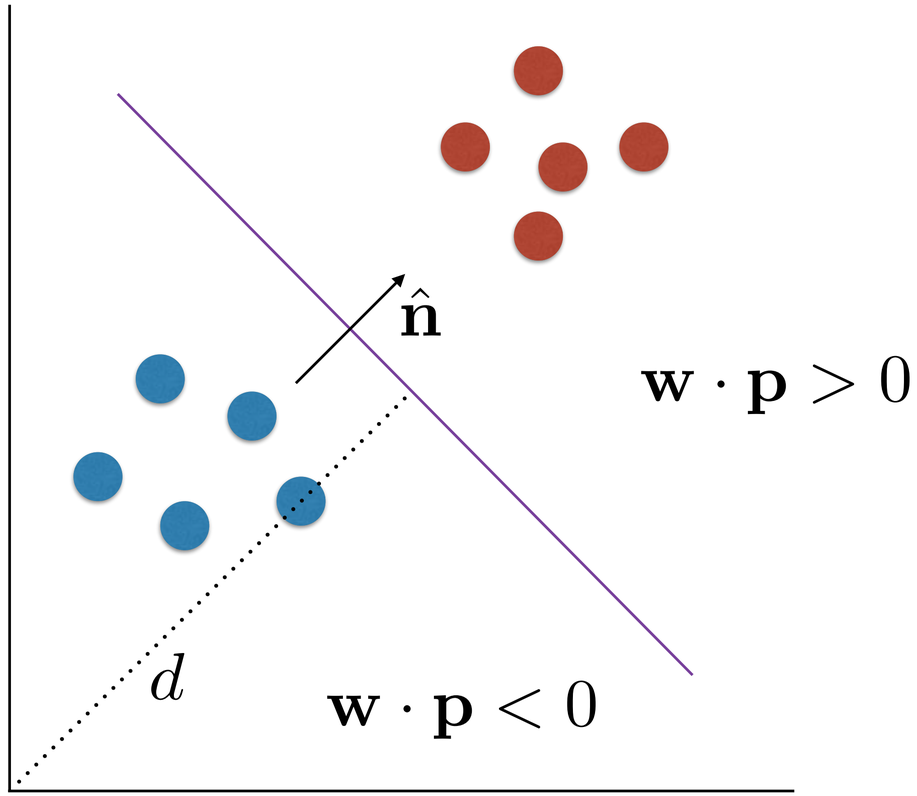
- Approach:
  - Measure “features” of the things
  - Put features together in a vector
  - Look at the problem geometrically
  - Changing the coordinate system can make a huge difference!

(basis for pattern recognition, machine learning, other AI, …)

Classification

$$\mathbf{p} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ 1 \end{bmatrix}$$

$$\mathbf{w} = \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_k \\ -d \end{bmatrix}$$



Classification

- But what if it's complicated, and you can't easily solve for **w**?
- Can you iteratively tweak the values in **w** until you "get it right"?
- The entries of **w** act as "weights" in a weighted combination of features, so it's called a *weight vector*

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \\ w_{k+1} \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ 1 \end{bmatrix}$$

$$\mathbf{w} \cdot \mathbf{p} = \sum_{i=1}^{k} w_i \; x_i + w_{k+1}$$

This is basically the heart of what neural networks do!

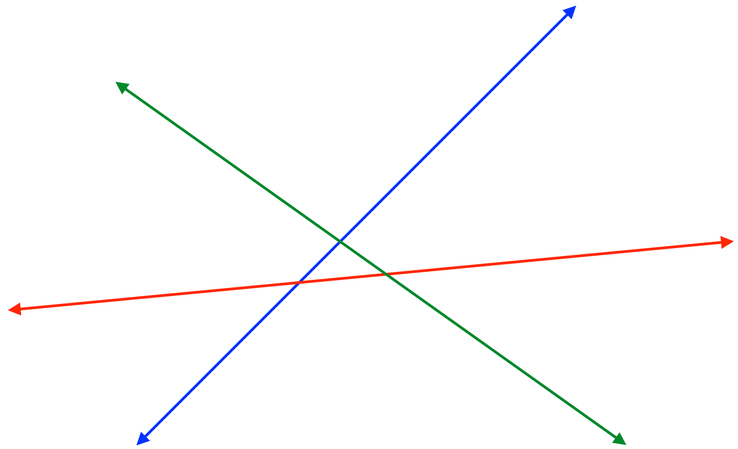
Systems of Equations

- If you need to solve for *n* unknowns, you need *n* equations, right?
- What if the data is noisy?
- Idea: get more data and let the noise "average out"
- But now there are too many equations!

$$\mathbf{Ax} = \mathbf{b}$$

Overconstrained Systems

- When you have too many equations, there may not be a solution
- Idea: get as close as possible to fitting all of the equations
- If you use a squared-error metric, this leads to a *least-squares solution*



$$\text{minimize} \quad \|\mathbf{Ax} - \mathbf{b}\|^2$$

Eigensystems

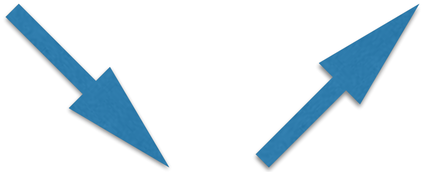
- The actions of some matrices are sometimes described more easily when converted to another coordinate system
- Some matrices are *pure scaling* along certain key directions
- A useful tool for analyzing these are *eigensystems*
  - Eigenvectors - directions of scaling
  - Eigenvalues - amount of scaling in each direction

Eigensystems

What does this matrix do?

$$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

Scales by 4 in the direction [1,1]
and by 2 in the direction [-1,1]



Eigenvectors:

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

Eigenvalues:

$$4, 2$$

Coming up...

- Applications in Computer Vision (and CS 450) — Dr. Farrell
- Applications in Computer Graphics (and CS 455) — Dr. Egbert