Final Review

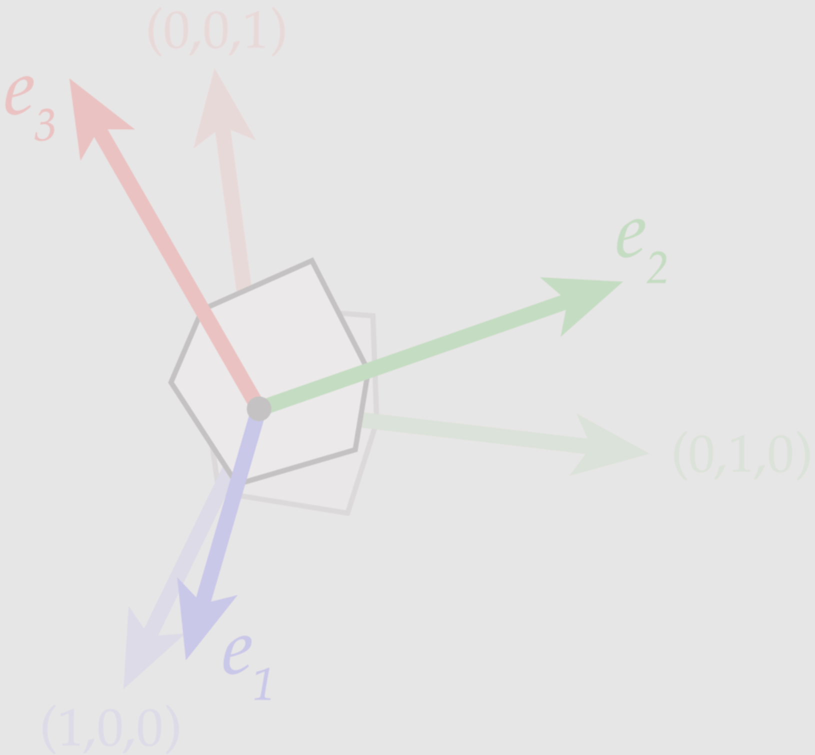
Final Overview

- 80 minutes, 5 Problems worth 125 points in total
    - **Will be graded out of 100 points** (anything after that is extra credit)
        - First 4 problems (100pts) are based on lecture material found in these review slides
        - 5th problem (extra 25pts) may or may not come from these review slides :)

- Cheat sheet: one 3x3 inch note (about the size of a post it note) front and back

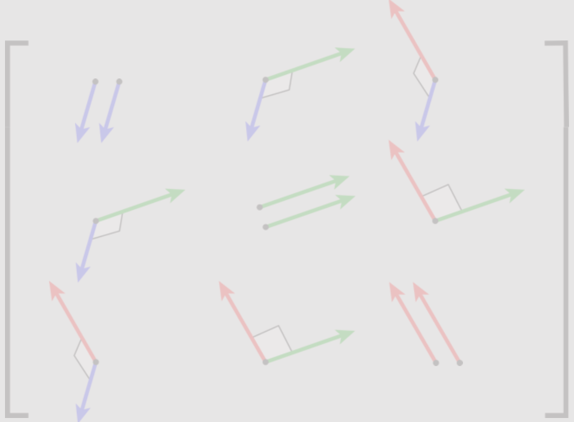
- **Please bring a pencil & pen to write your solutions**

3D Inverse Rotations



$$\begin{array}{cc} R^\top & R \\ \begin{bmatrix} - & e_1^T & - \\ - & e_2^T & - \\ - & e_3^T & - \end{bmatrix} & \begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix} \end{array}$$

$=$



$$= \begin{bmatrix} e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\ e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\ e_3^T e_1 & e_3^T e_2 & e_3^T e_3 \end{bmatrix}$$

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

$$R^T R = I \;\Rightarrow\; R^T = R^{-1}$$

If you need to review any slides more in depth, look here for which lecture it came from

- A1: Rasterization
- A2: Geometry
- A3: Rendering
- A4: Animation

Pixel Pushing

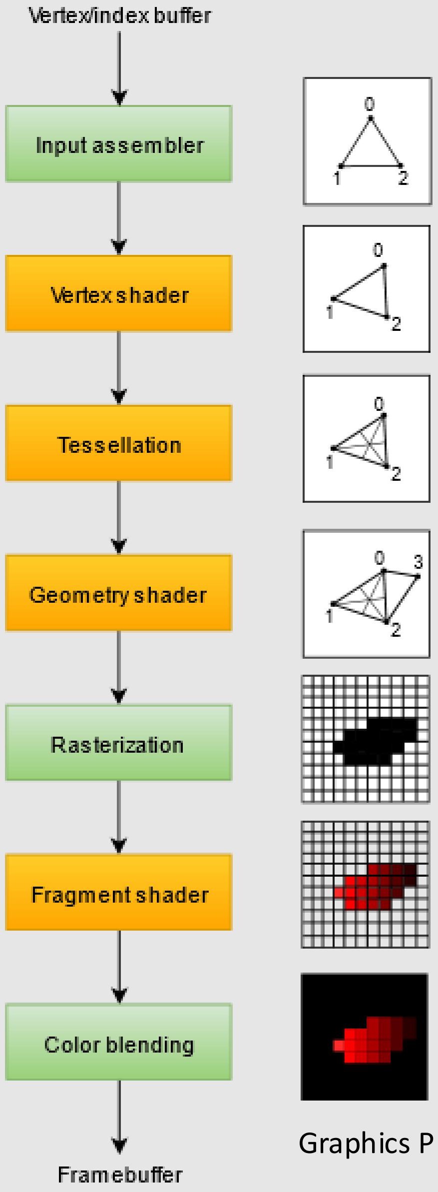
- Shaders
    - Vertex Shader
    - Fragment Shader

- Transformations
    - Translate
    - Scale
    - Rotate

- Perspective Transform

- Scene Graphs

The Graphics Pipeline

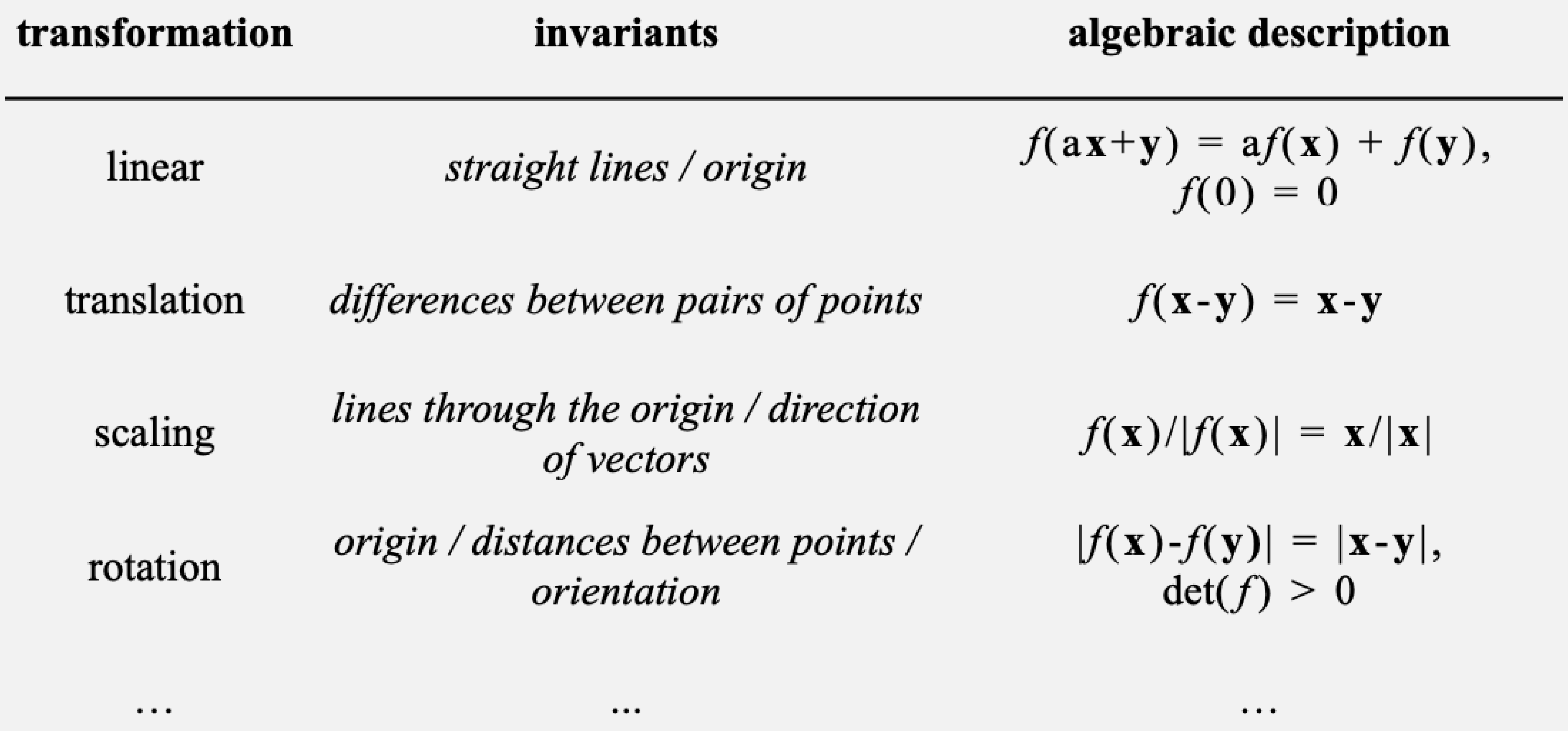


Graphics Pipeline Tutorial (2019) Vulkan

- Sometimes called the:
  - 3D Graphics Pipeline
  - Rasterization Pipeline
  - GPU Pipeline
- GPU was designed specifically to run this pipeline fast
- Entire pipeline was fixed-function.
  - You provide the data, a vertex shader, and a fragment shader, and the GPU does the rest.
  - Fixed-function == fast!
    - By limiting what an architecture can do, that makes the architecture really good at what it can do.
      - In graphics, we need to run the same operations over millions of datapoints.

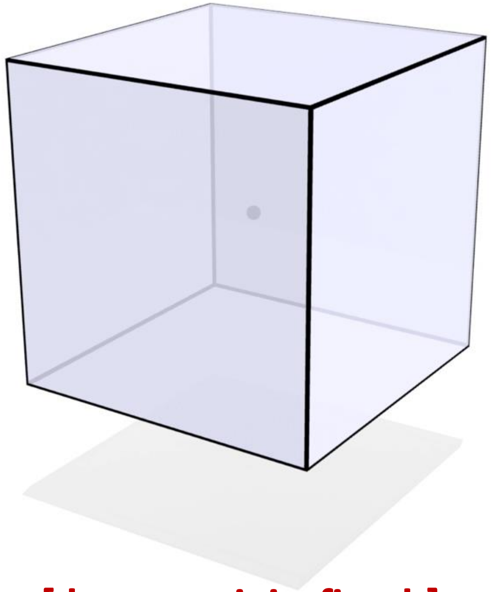
Invariants of Transformation

A transformation is determined by the **invariants** it preserves

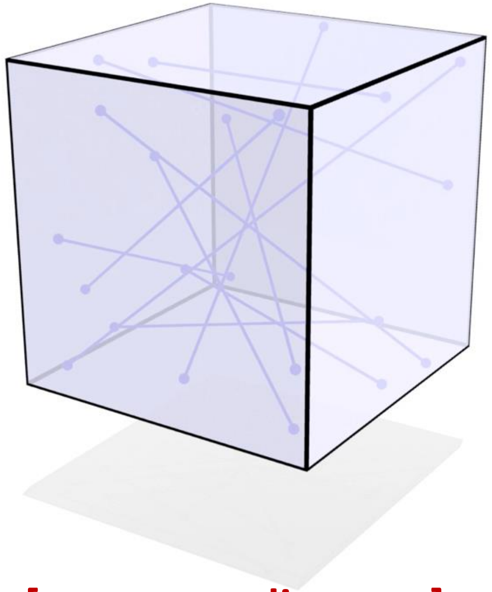


transformation	invariants	algebraic description
linear	*straight lines / origin*	$f(a\mathbf{x}+\mathbf{y}) = af(\mathbf{x}) + f(\mathbf{y})$, $f(0) = 0$
translation	*differences between pairs of points*	$f(\mathbf{x}-\mathbf{y}) = \mathbf{x}-\mathbf{y}$
scaling	*lines through the origin / direction of vectors*	$f(\mathbf{x})/\lvert f(\mathbf{x})\rvert = \mathbf{x}/\lvert\mathbf{x}\rvert$
rotation	*origin / distances between points / orientation*	$\lvert f(\mathbf{x})-f(\mathbf{y})\rvert = \lvert\mathbf{x}-\mathbf{y}\rvert$, $\det(f) > 0$
…	…	…

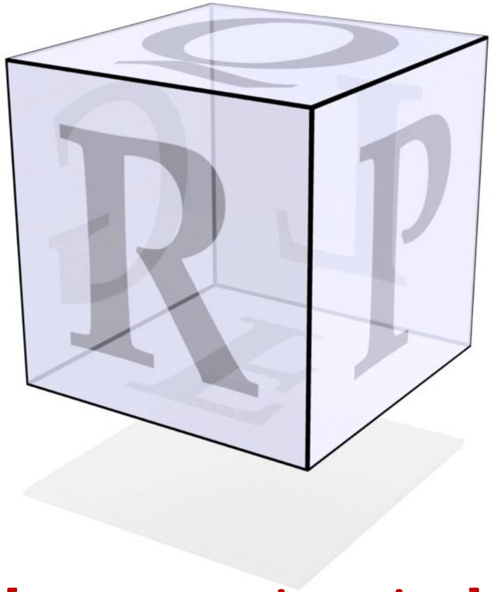
Rotation



[keeps origin fixed]



[preserves distance]



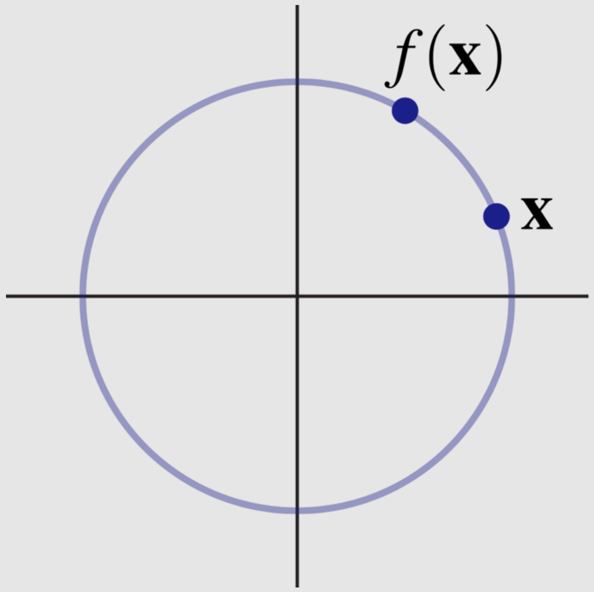
[preserves orientation]

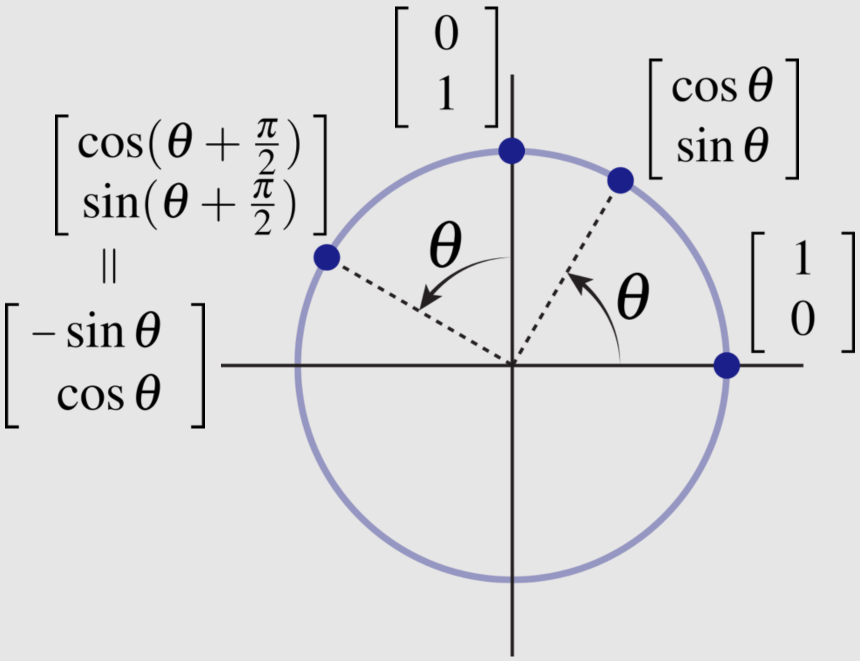
First two properties imply rotations are **linear**

We say that a transform preserves orientation if $\det(T) > 0$

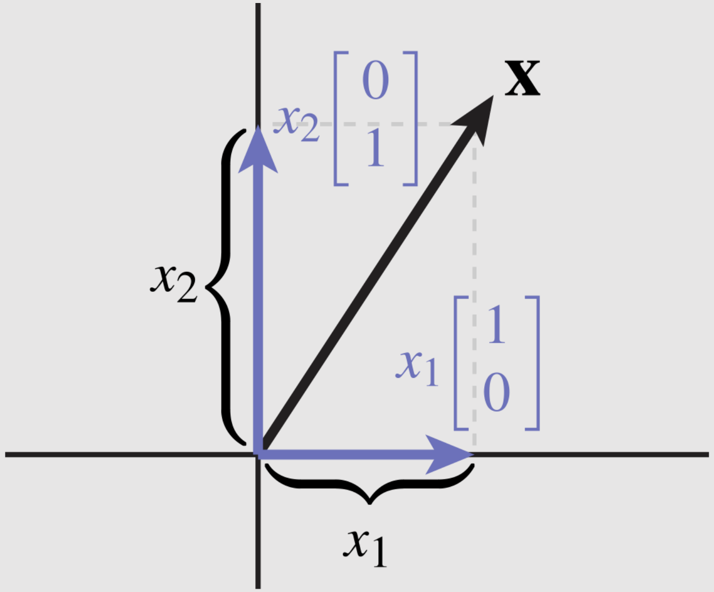
2D Rotations

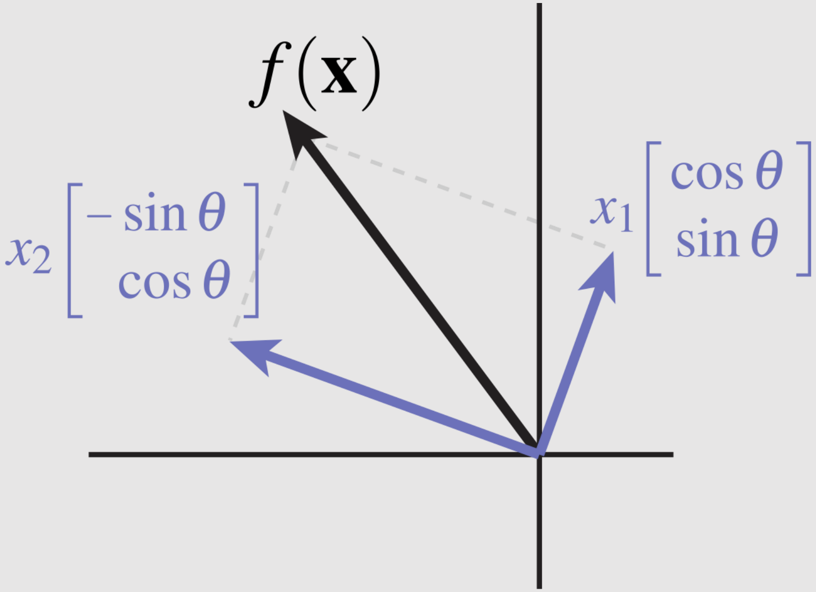
Rotations preserve distances and the origin—hence, a 2D rotation by an angle θ maps each point x to a point $f(x)$ on the circle of radius $|x|$:





2D Rotations





$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$f(\mathbf{x}) = x_1 \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} + x_2 \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$$

Rotations (like all transforms) are linear maps.
We can express the transform as a change of bases:

$$f_\theta(\mathbf{x}) = \begin{bmatrix} \cos\theta & -\sin(\theta) \\ \sin\theta & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Reflections

- Does every matrix $Q^{\mathsf{T}}Q = I$ represent a rotation?
  - Must preserve:
    - Origin
    - Distance
    - Orientation

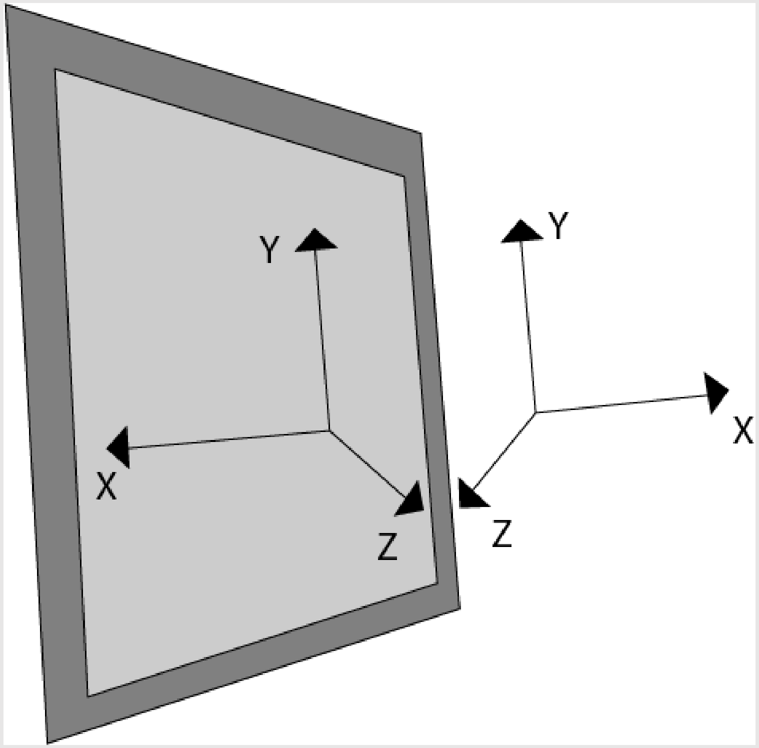
- Consider:

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

- Just like rotations, Q has nice inverse properties:

$$Q^{\mathsf{T}}Q = \begin{bmatrix} (-1)^2 & 0 \\ 0 & 1 \end{bmatrix} = I$$

- But the determinant is **negative!**
  - Not orientation preserving



Scaling

- Each vector u gets scaled by some scalar a

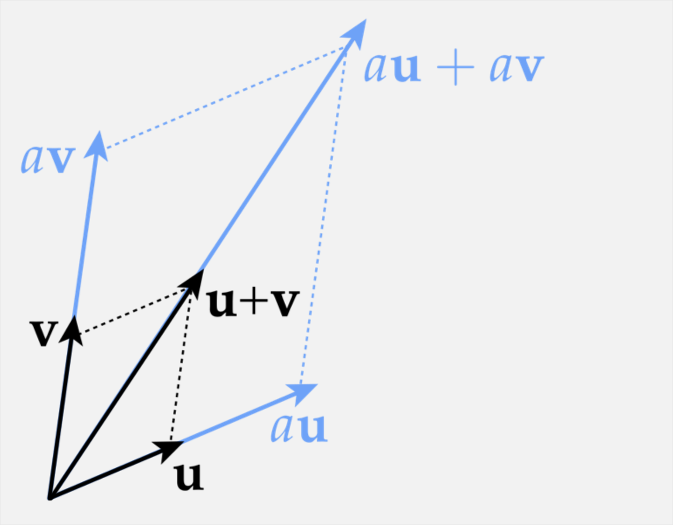
$$f(\mathbf{u}) = a\mathbf{u}, a \in \mathbb{R}$$

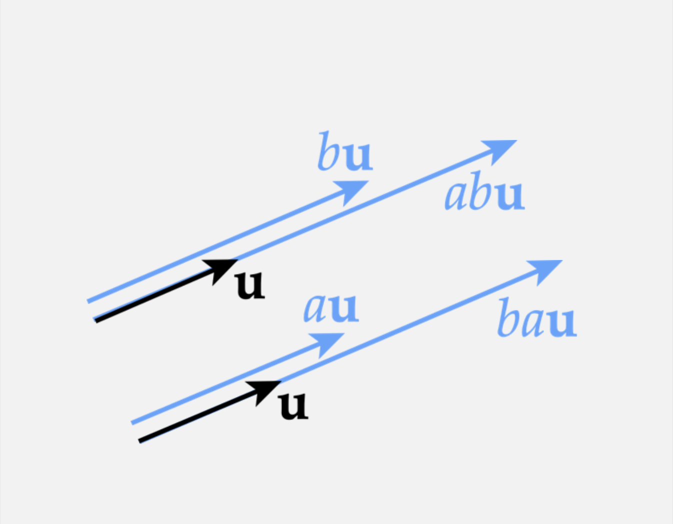
- Scaling is a linear transformation
    - Multiplication:

$$f(b\mathbf{u}) = ab\mathbf{u} = ba\mathbf{u} = bf(\mathbf{u})$$

    - Addition:

$$\begin{gathered} f(\mathbf{u}+\mathbf{v}) = \\ a(\mathbf{u}+\mathbf{v}) = \\ a\mathbf{u} + a\mathbf{v} = \\ f(\mathbf{u}) + f(\mathbf{v}) \end{gathered}$$





Negative Scaling

Can think of negative scaling as a series of reflections

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

Also works in 3D:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \underset{\text{[flip x]}}{\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}} \underset{\text{[flip y]}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}} \underset{\text{[flip z]}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}}$$

In 2D, reflection reverses orientation twice ($\det(T) > 0$)
In 3D, reflection reverses orientation thrice ($\det(T) < 0$)

Non-Uniform Scaling

- To scale a vector u by a non-uniform amount (a, b, c):

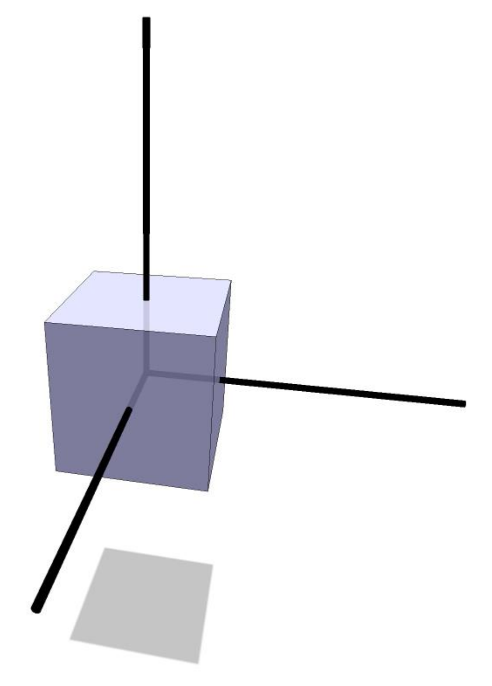
$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} au_1 \\ bu_2 \\ cu_3 \end{bmatrix}$$

- The above works only if scaling is axis-aligned. What if it isn't?
- Idea:
  - Rotate to a new axis R
  - Perform axis-aligned scaling D
  - Rotate back to original axis R^T

$$A := R^T D R$$

  - Resulting transform A is a symmetric matrix

- **Q:** Do all symmetric matrices represent non-uniform scaling?



Shear

- A shear displaces each point x in a direction u according to its distance along a fixed vector v:

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

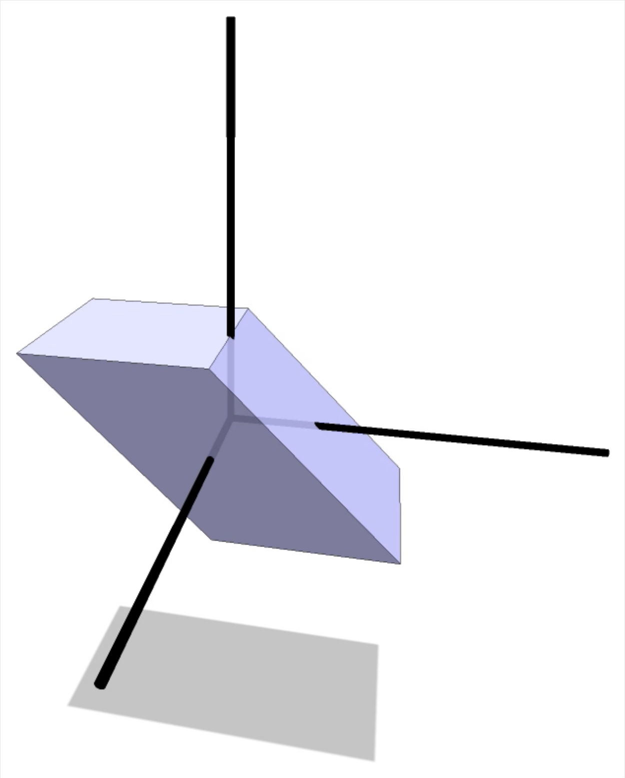
- Still a linear transformation—can be rewritten as:

$$A_{\mathbf{u},\mathbf{v}} = I + \mathbf{u}\mathbf{v}^{\mathsf{T}}$$

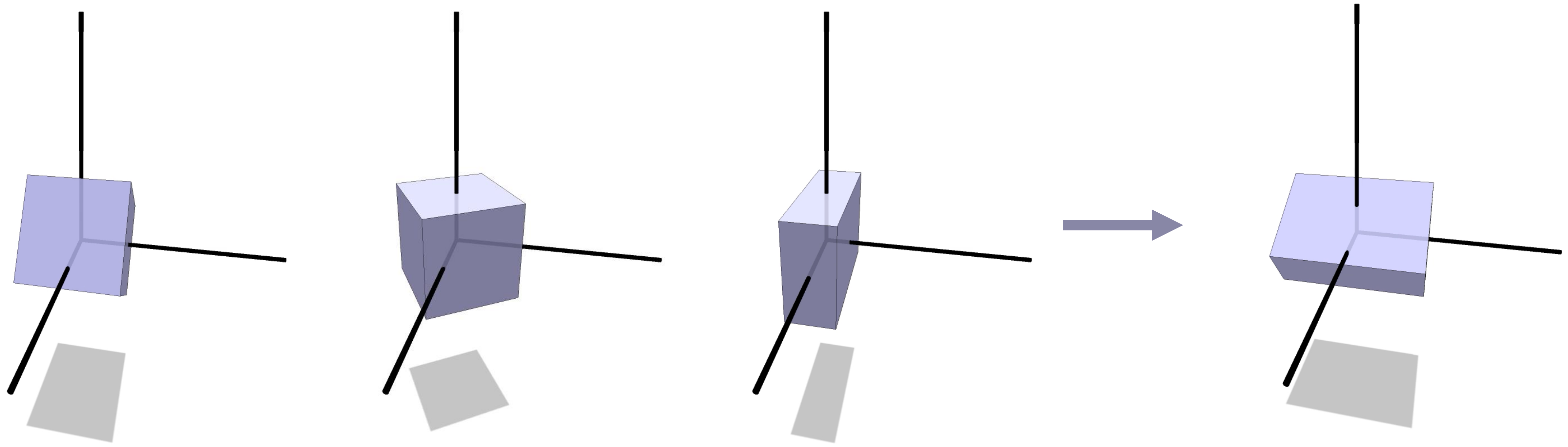
- Example:

$$\mathbf{u} = (\cos(t), 0,0)$$
$$\mathbf{v} = (0,1,0)$$

$$A_{\mathbf{u},\mathbf{v}} = \begin{bmatrix} 1 & \cos(t) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Composing Transforms

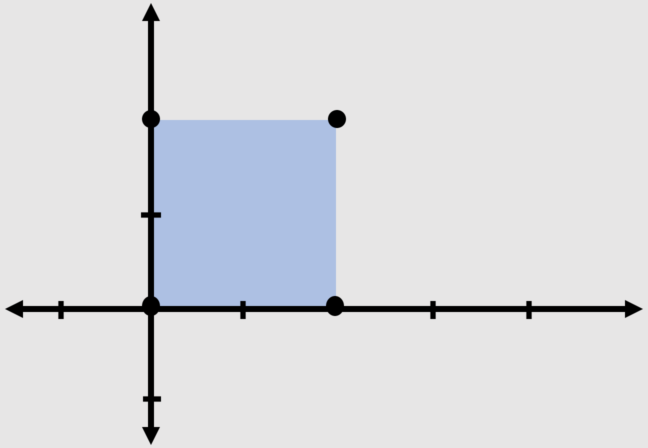


$R_x(t)$ $R_y(t)$ $S(t)$ $A(t) = R_x(t)R_y(t)S(t)$

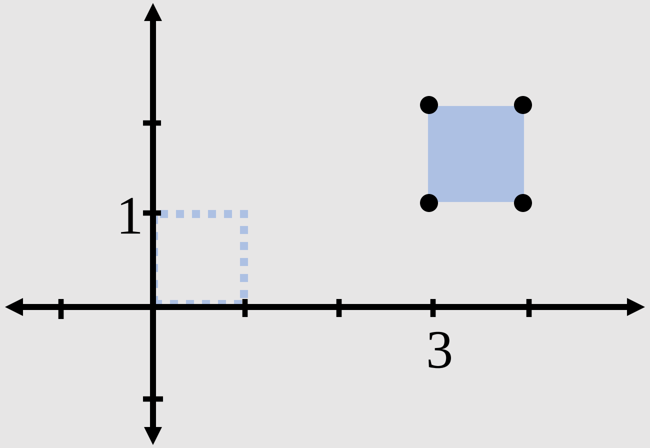
We can now build up composite transformations via matrix multiplication

Composing Transforms

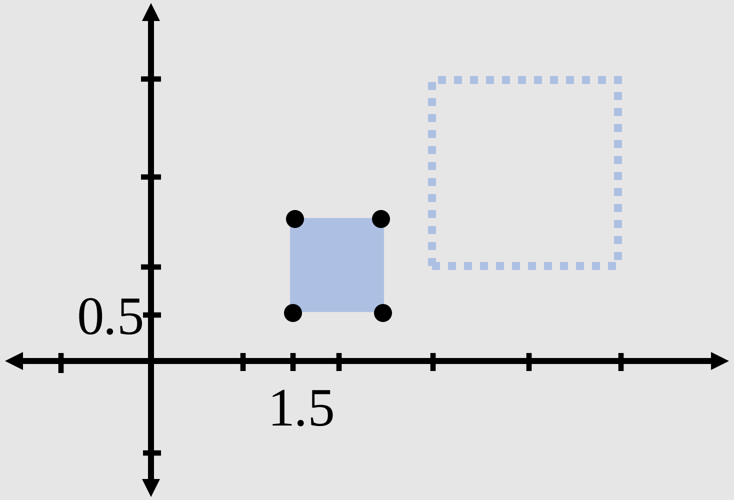
- Order matters when compositing transforms!



[original]



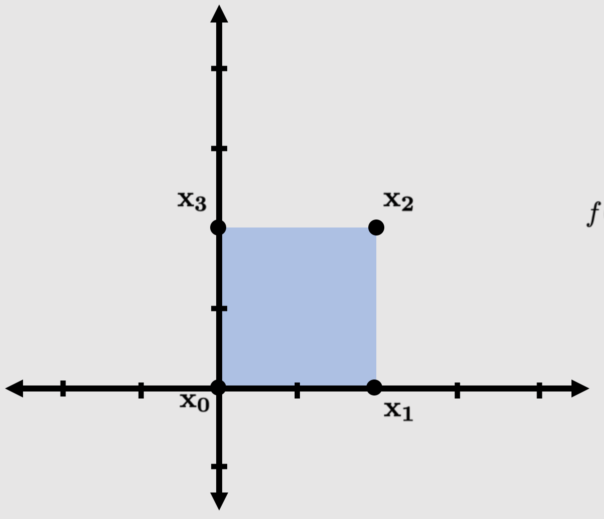
[scale by 1/2, then translate by (3,1)]

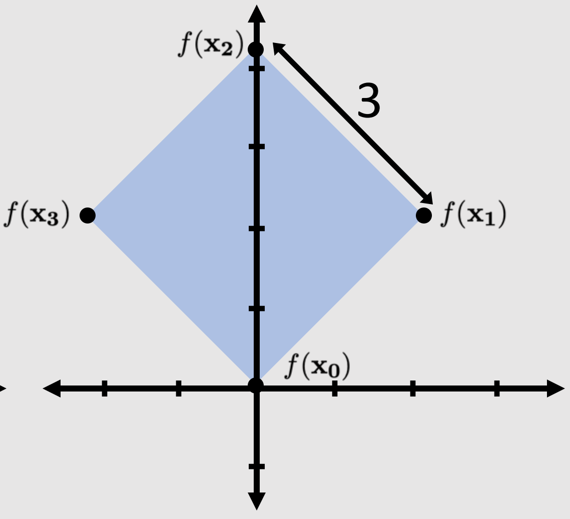


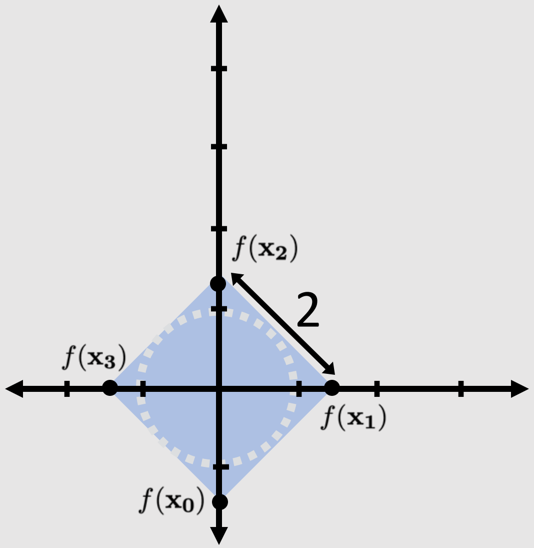
[translate by (3,1), then scale by 1/2]

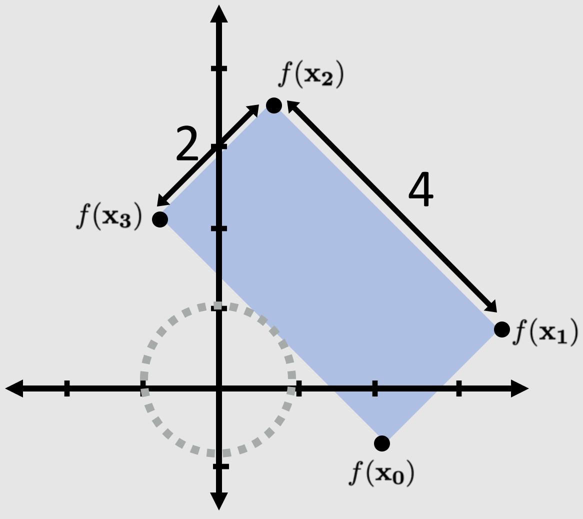
Composing Transforms

How would you perform these transformations?**



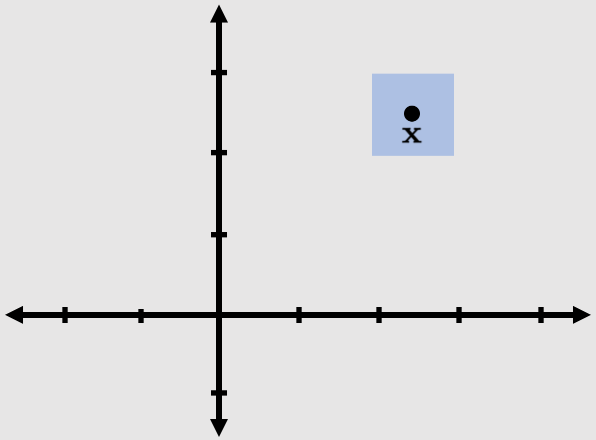




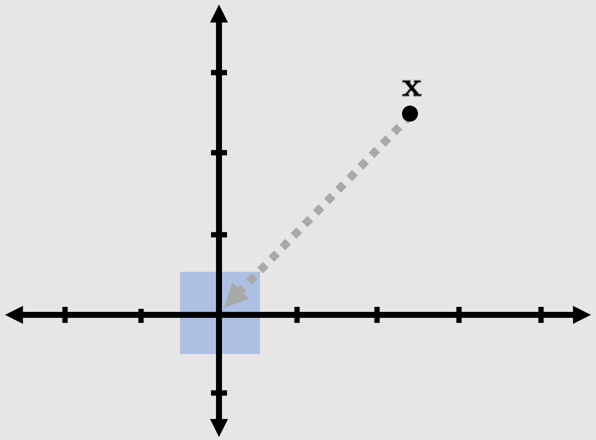


**remember there's always more than one way to do so

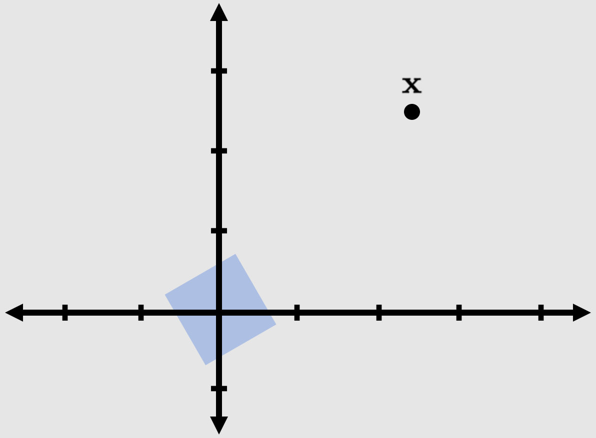
Rotating About A Point



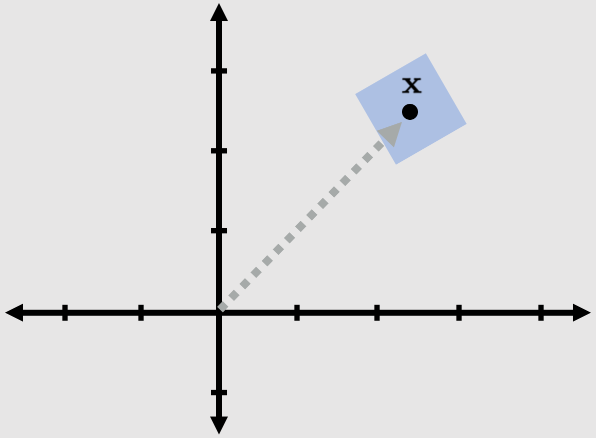
[Step 0] compute x (dist. from origin)



[Step 1] translate by -x



[Step 2] rotate



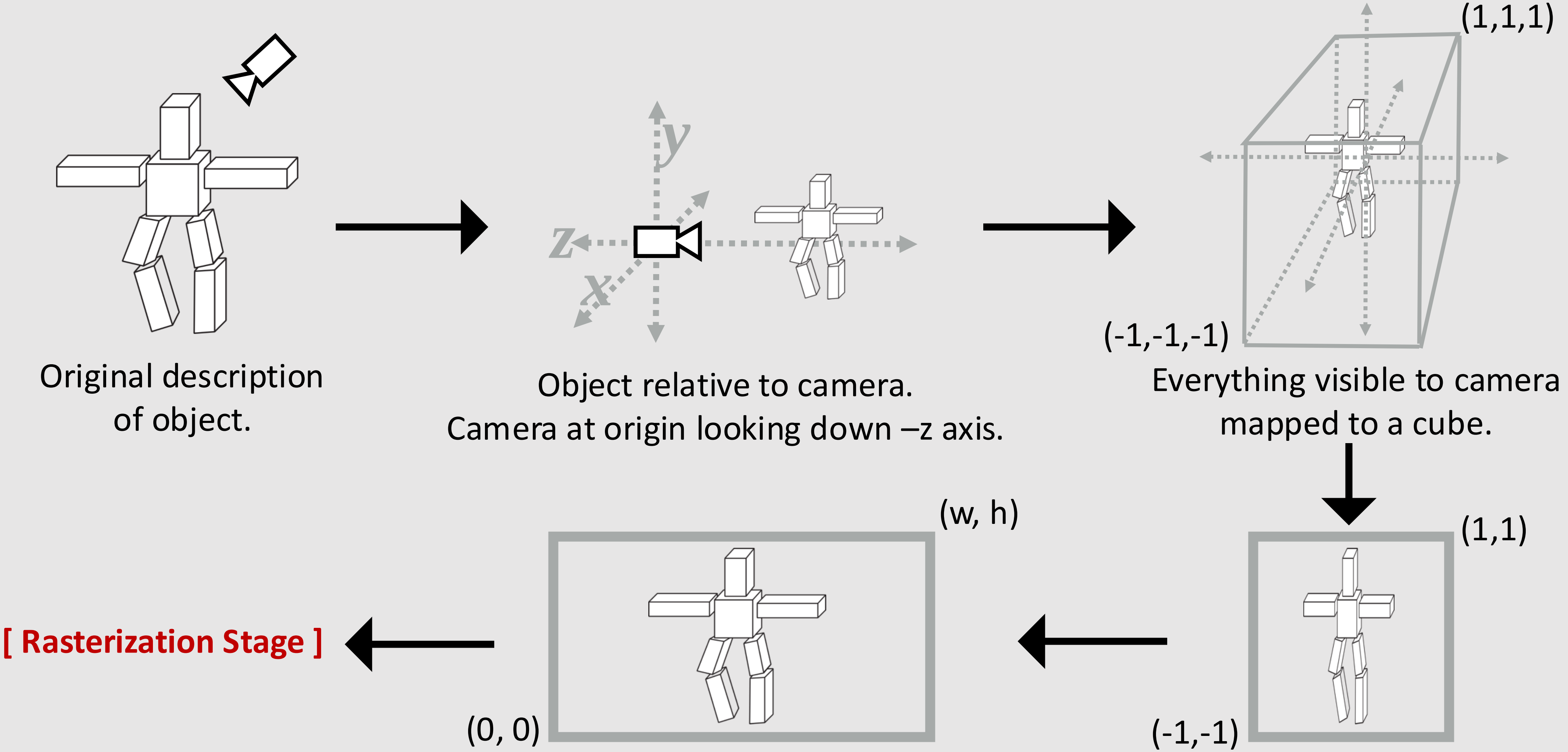
[Step 3] translate by x

Perspective Projection

parallel lines converge at the horizon

distant objects appear smaller

Perspective Projection



Original description of object.

Object relative to camera.
Camera at origin looking down –z axis.

Everything visible to camera mapped to a cube.

Everything visible to camera mapped to a cube.

Coordinates stretched to image dims.
Image flipped upside down.

[Rasterization Stage]

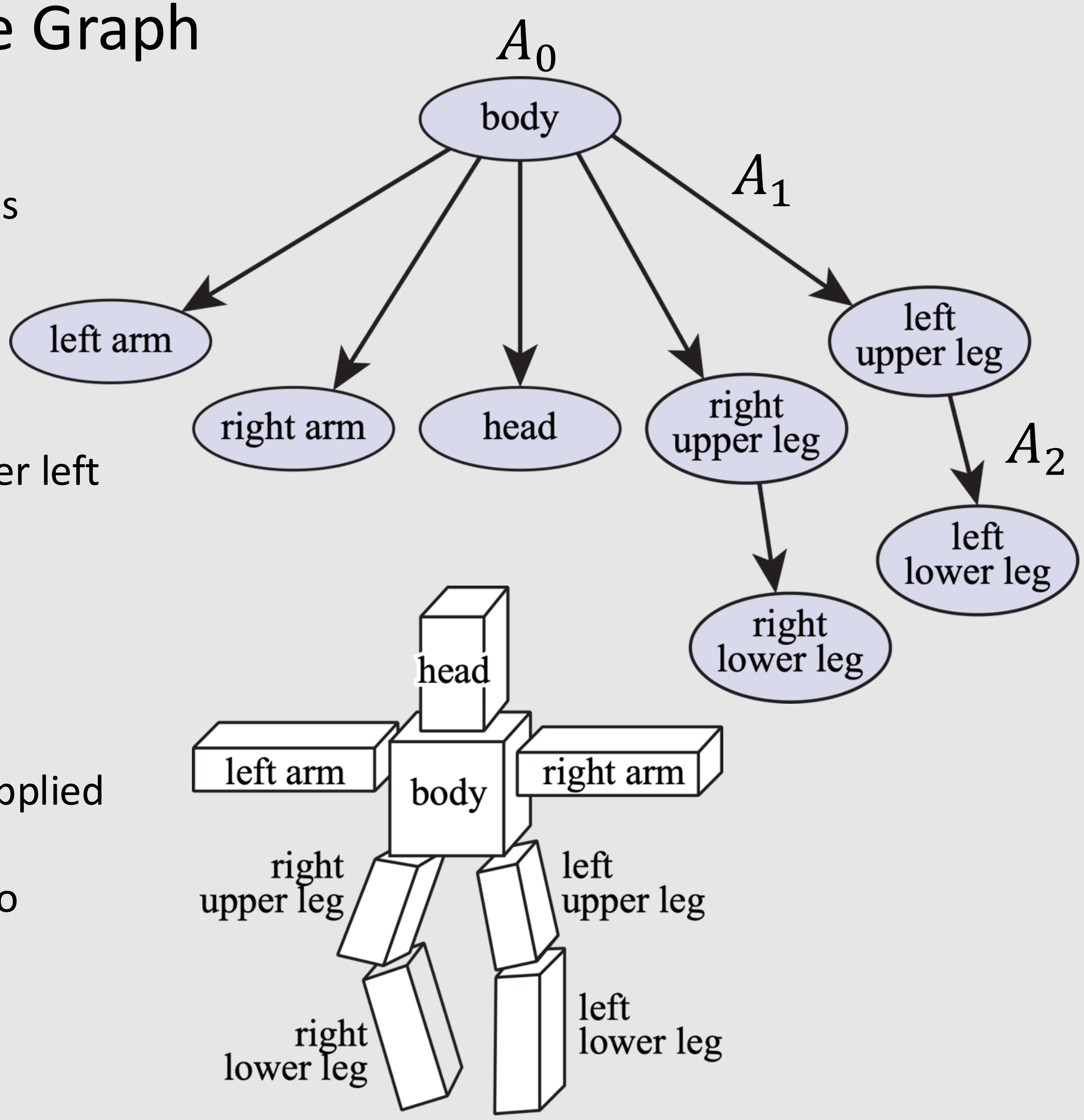
Scene Graph

- Suppose we want to build a skeleton out of cubes

- **Idea:** transform cubes in world space
  - Store transform of each cube

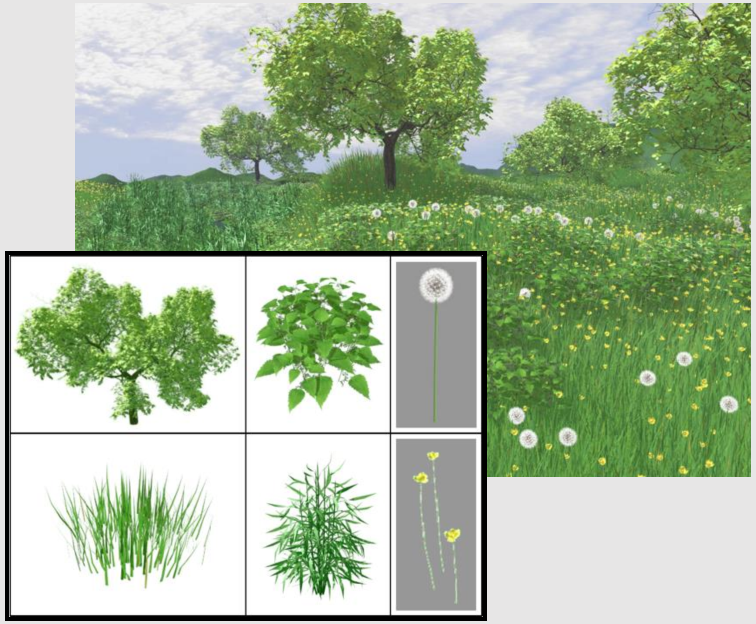
- **Problem:** If we rotate the left upper leg, the lower left leg won’t track with it
  - **Better Idea:** store a hierarchy of transforms
    - Known as a scene graph
    - Each edge (+root) stores a linear transformation
    - Composition of transformations gets applied to nodes
      - Keep transformations on a stack to reduce redundant multiplication

- **Lower left leg transform:** $A_2 A_1 A_0$

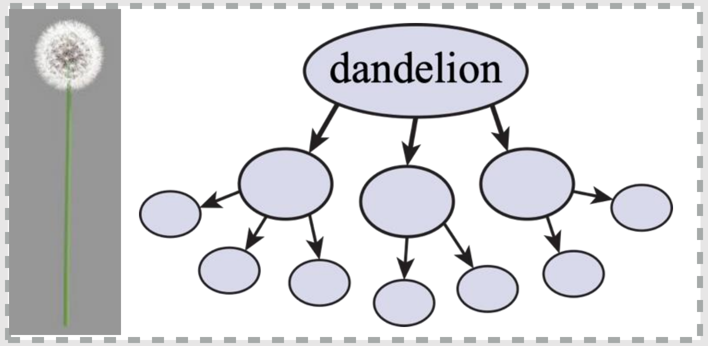


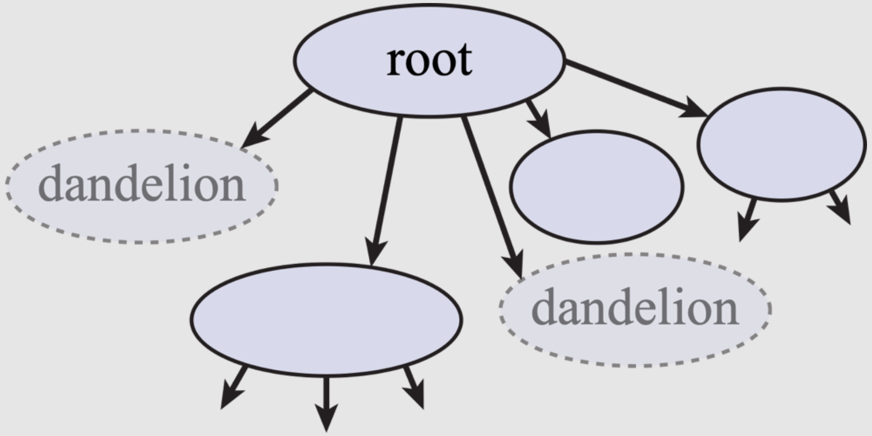
Instancing

- What if we want many copies of the same object in a scene?
    - Rather than have many copies of the geometry, scene graph, we can just put a "pointer" node in our scene graph
        - Saves a reference to a shared geometry
        - Specify a transform for each reference
            - Careful: Modifying the geometry will modify all references to it



Realistic modeling and rendering of plant ecosystems (1998) Deussen et al





- ~~A1: Rasterization~~
- A2: Geometry
- A3: Rendering
- A4: Animation

Meshes

- Types of Geometric Representations
  - Algebraic Surfaces
  - CSG
  - Blobby
  - Level Set
  - Fractals
  - Point Cloud
  - Meshes

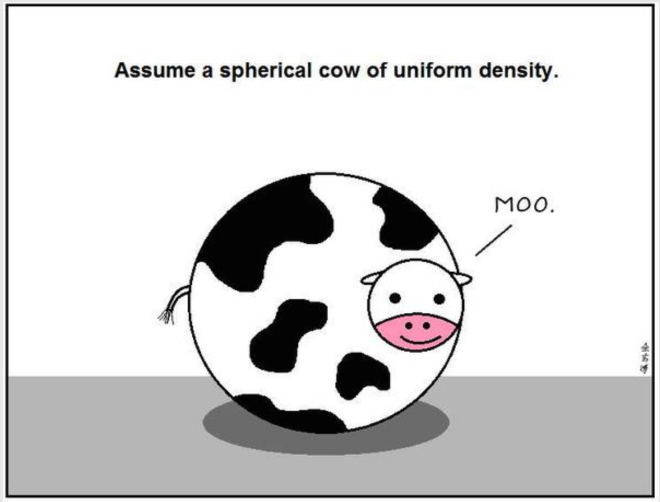
- Global Mesh Operations
  - Subdivision
  - Isotropic Remeshing

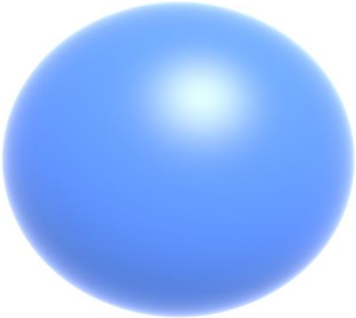
- Spatial Data Structures
  - BVH
  - KD-Tree
  - Uniform Grid
  - Quadtree/Octree

Algebraic Surfaces [Implicit]

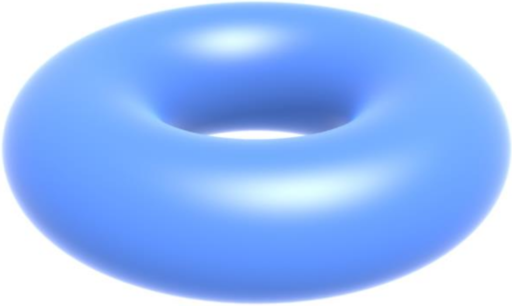
- Simple way to think of it: a surface built with algebra [math]
  - Intuitively thought of as a surface where points are some radius r away from another point/line/surface

- Easy to generate smooth/symmetric surfaces
  - Difficult to generate impurities/deformations





$$x^2+y^2+z^2=1$$

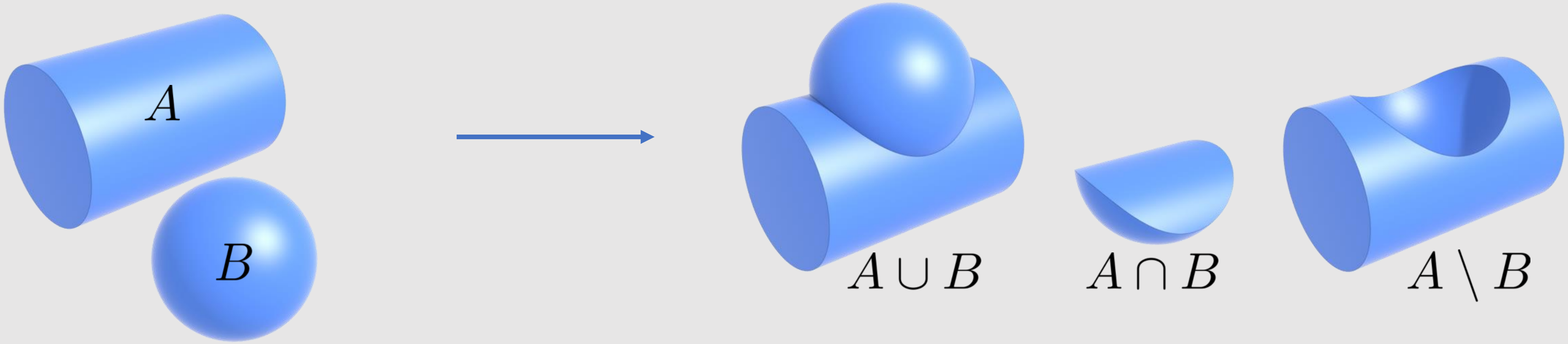


$$(R-\sqrt{x^2+y^2})^2+z^2=r^2$$

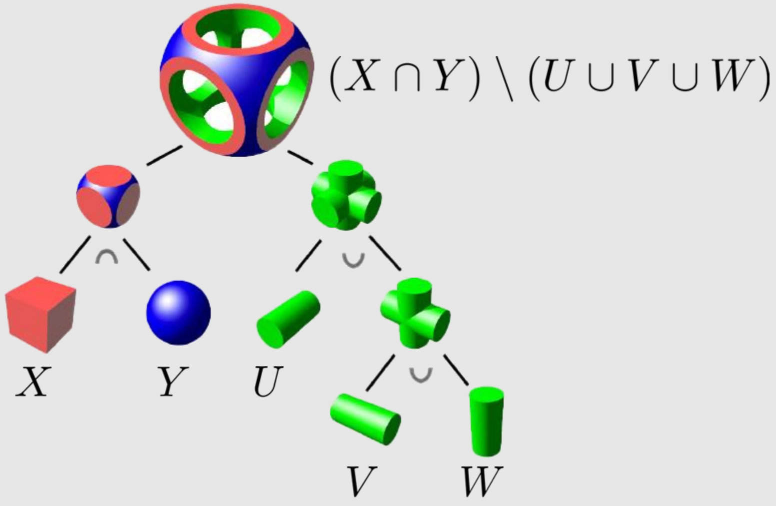
$$(x^2+\frac{9y^2}{4}+z^2-1)^3 = x^2z^3+\frac{9y^2z^3}{80}$$

Constructive Solid Geometry [Implicit]

- Build more complicated shapes via Boolean operations
  - Basic operations:

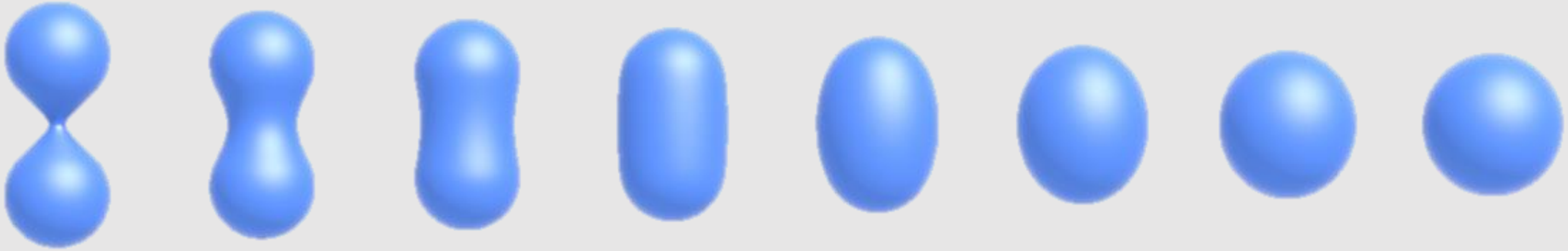


- Can be used to form complex shapes!



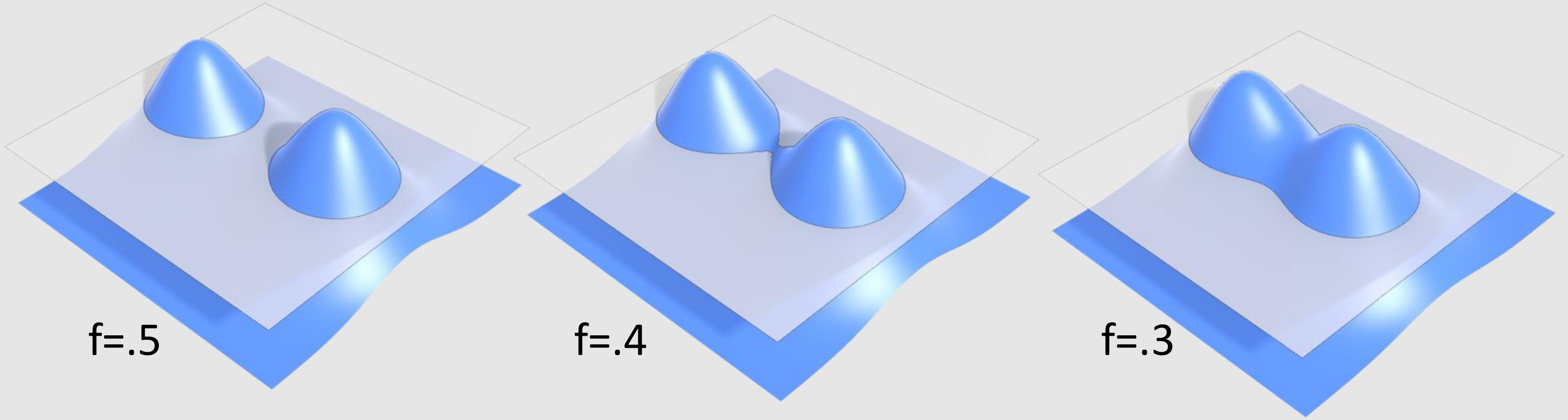
Blobby Surfaces [Implicit]

- Instead of Booleans, gradually blend surfaces together:



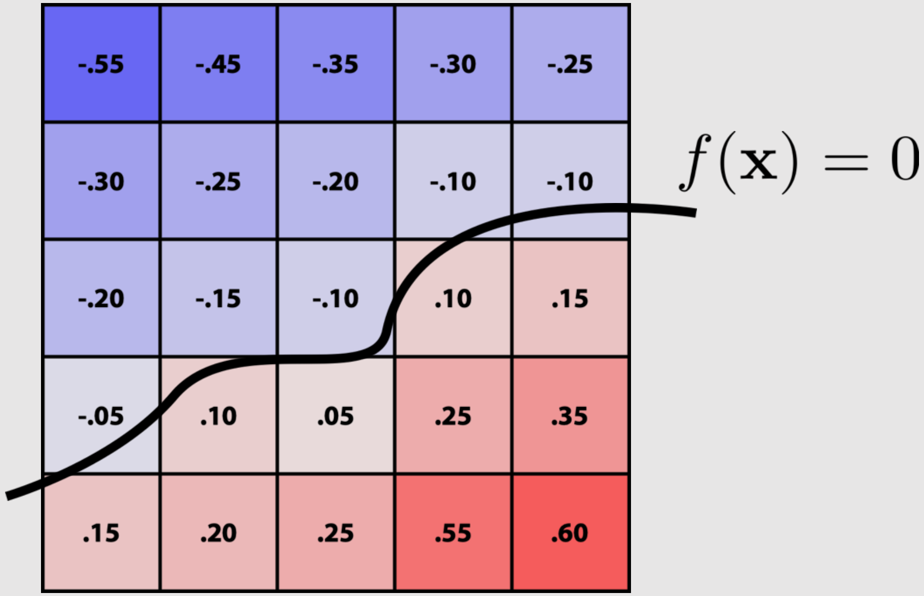
- Easier to understand in 2D: $\phi_p(x) := e^{-|x-p|^2}$ (Gaussian centered at p)

$f := \phi_p + \phi_q$ (Sum of Gaussians centered at different points)

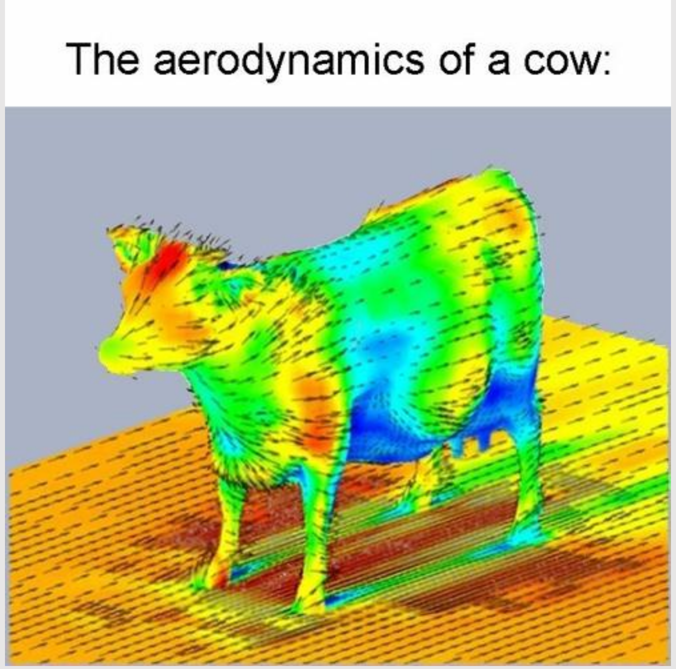


Level Set Methods [Implicit]

- Implicit surfaces have some nice features (e.g., merging/splitting)
  - But, hard to describe complex shapes in closed form
  - **Alternative:** store a grid of values approximating function



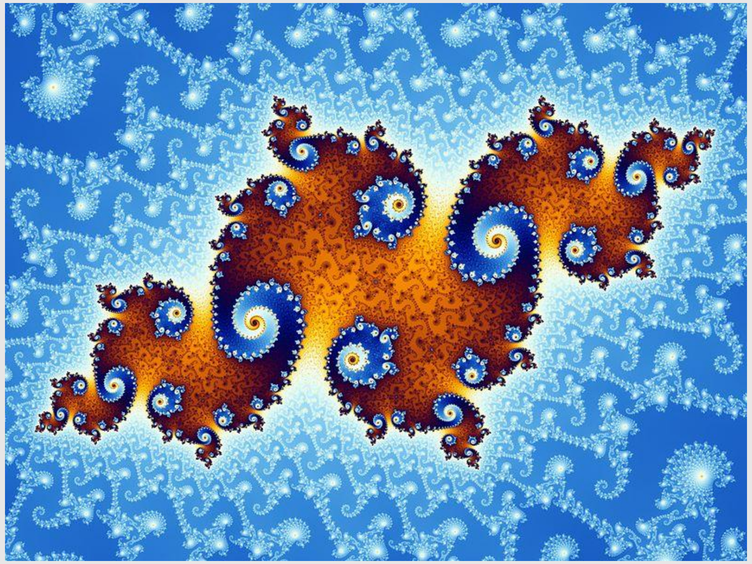
-.55	-.45	-.35	-.30	-.25
-.30	-.25	-.20	-.10	-.10
-.20	-.15	-.10	.10	.15
-.05	.10	.05	.25	.35
.15	.20	.25	.55	.60


The aerodynamics of a cow:

- Surface is found where interpolated values equal zero
- Provides much more explicit control over shape (like a texture)
- Unlike closed-form expressions, runs into problems of aliasing!

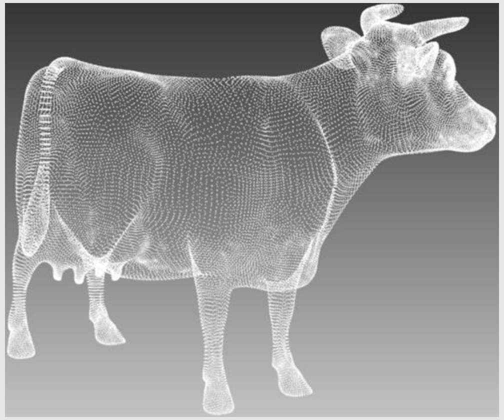
Fractals [Implicit]

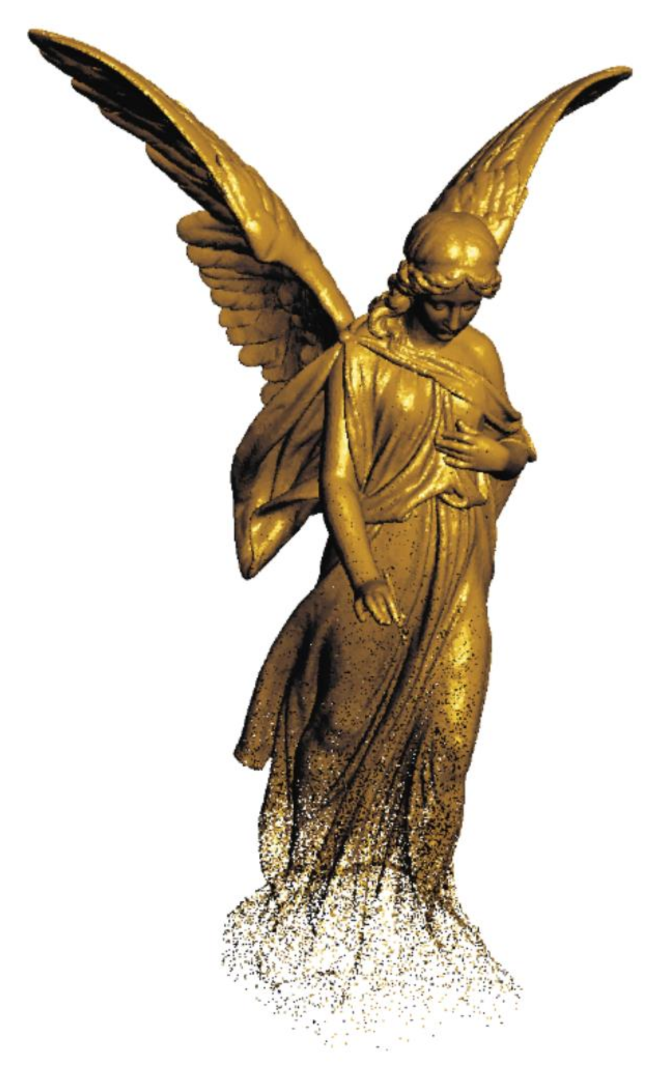
- No precise definition; exhibit self-similarity, detail at all scales
- New “language” for describing natural phenomena
- Hard to control shape!



Point Cloud [Explicit]

- Easiest representation: list of points (x, y, z)
  - Often augmented with normal
- Easily represent any kind of geometry
- Easy to draw dense cloud (>>1 point/pixel)
- Easy for simulating large deformation or topology changes, e.g. fluids, fracture
- Large lookup time
- Large memory overhead
  - Hard to interpolate undersampled regions
  - Slow to do processing / simulation /
  - Result is just as good as the scan

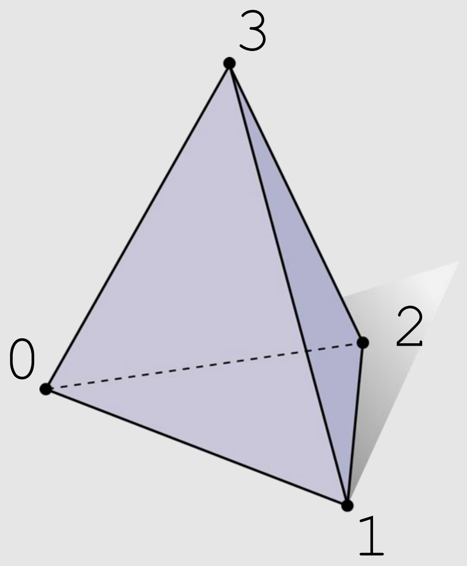


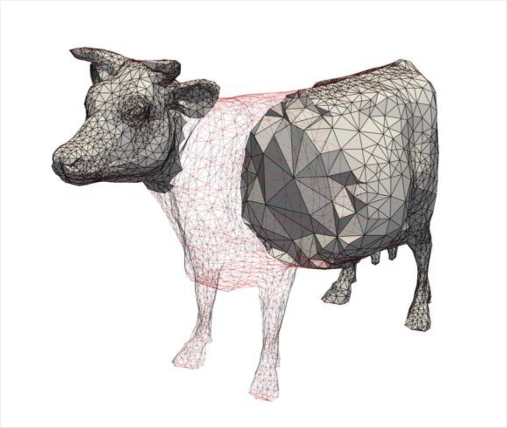


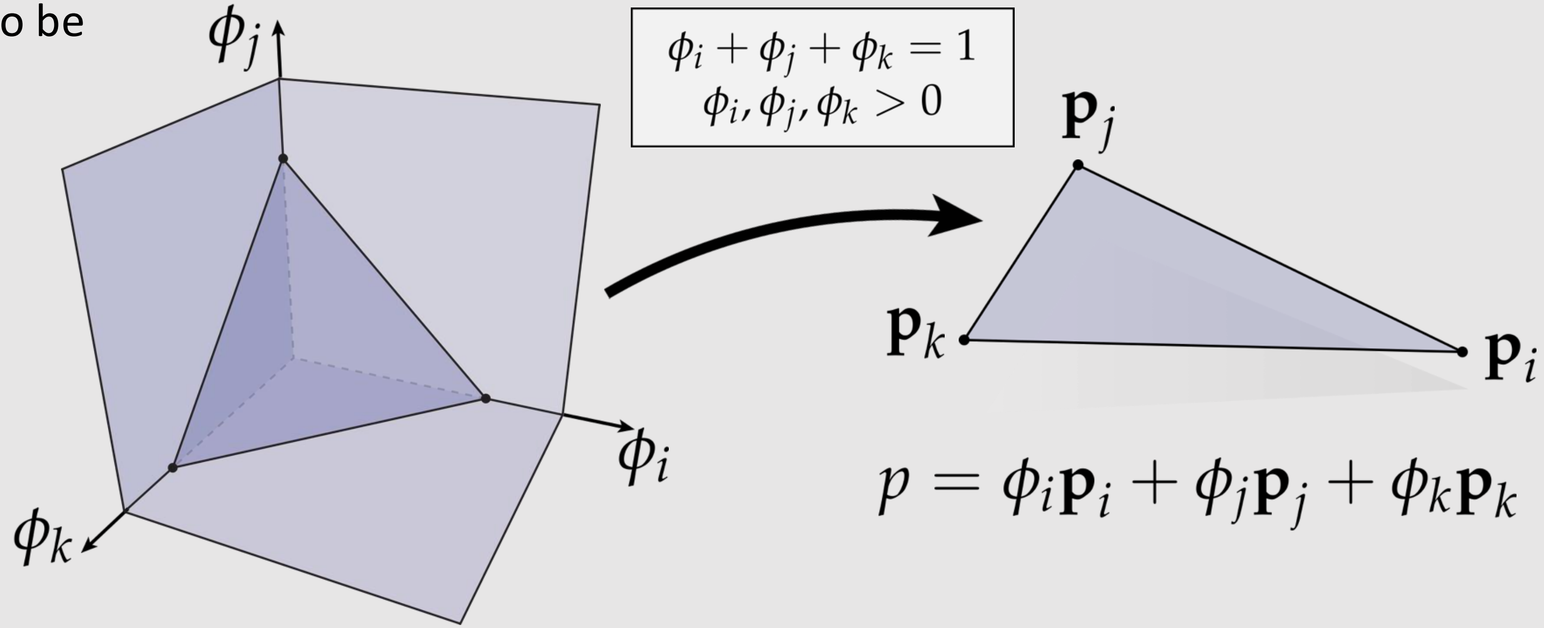
Triangle Mesh [Explicit]

- Larger memory overhead than point clouds
  - Store vertices as triples of coordinates *(x,y,z)*
  - Store triangles as triples of indices *(i,j,k)*
- Easy interpolation with good approximation
  - Use barycentric interpolation to define points inside triangles
- Polygonal Mesh: shapes do not need to be triangles
  - Ex: quads

	[VERTICES]			**[TRIANGLES]**		
	x	**y**	**z**	**i**	**j**	**k**
0:	-1	-1	-1	0	2	1
1:	1	-1	1	0	3	2
2:	1	1	-1	3	0	1
3:	-1	1	1	3	1	2







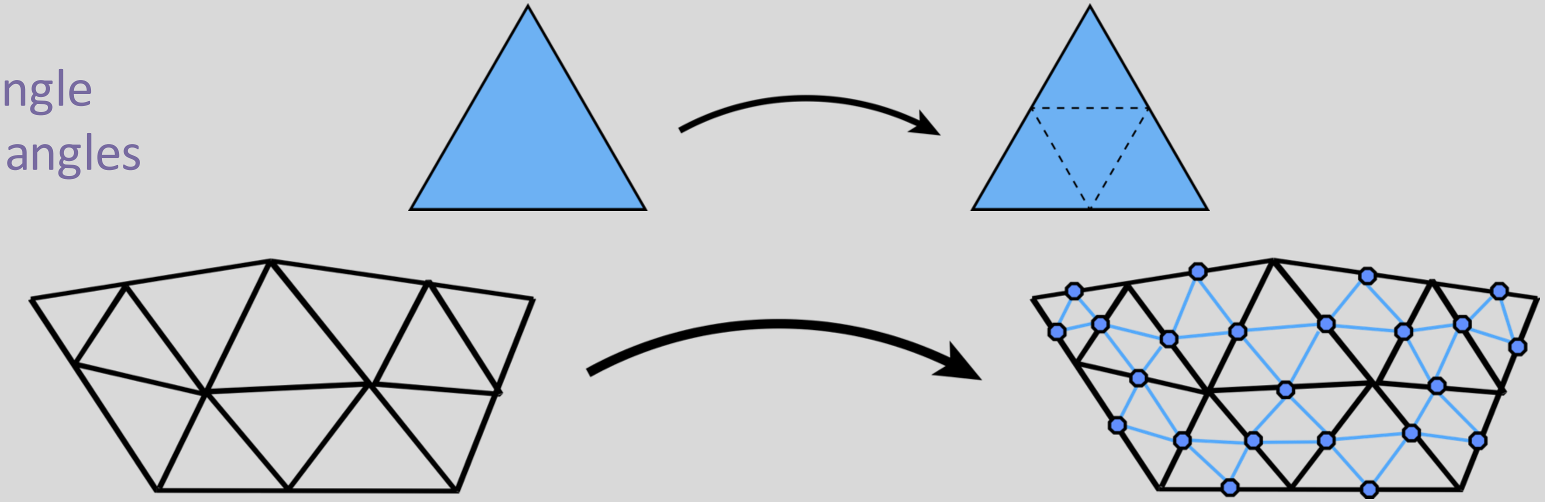
$$\phi_i + \phi_j + \phi_k = 1$$
$$\phi_i, \phi_j, \phi_k > 0$$

$$p = \phi_i \mathbf{p}_i + \phi_j \mathbf{p}_j + \phi_k \mathbf{p}_k$$

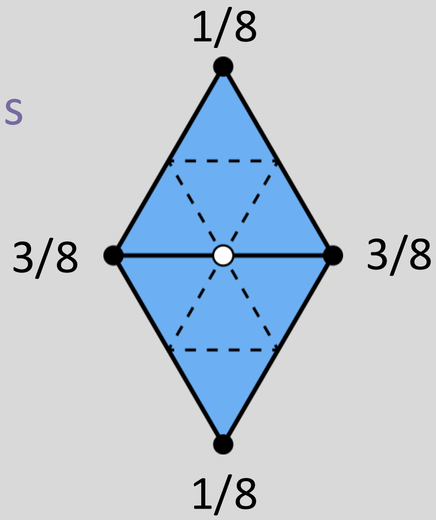
Loop Subdivision

Refine and upsample the mesh with additional smoothness.

Step 1:
Split triangle
into 4 triangles

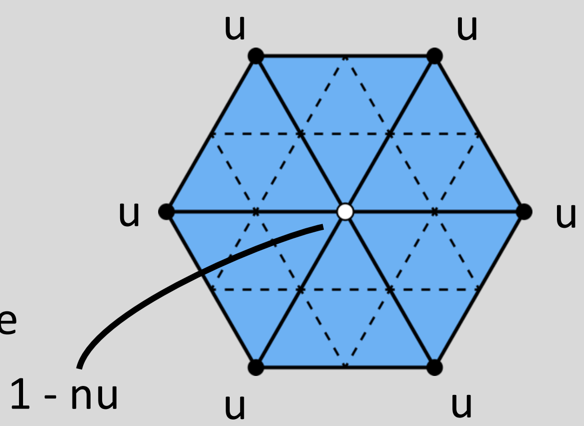


Step 2:
Assign new coords



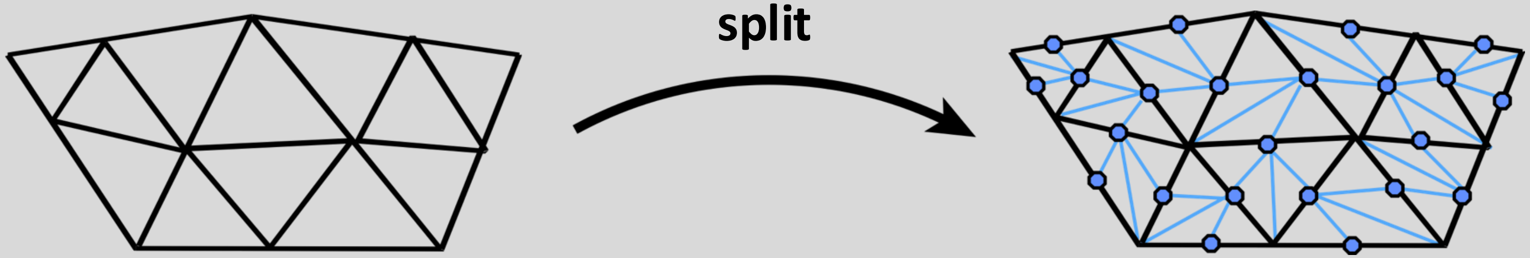
Step 3:
Assign old coords

n - vertex degree
u - 3/16 if n=3
3/(8n) otherwise

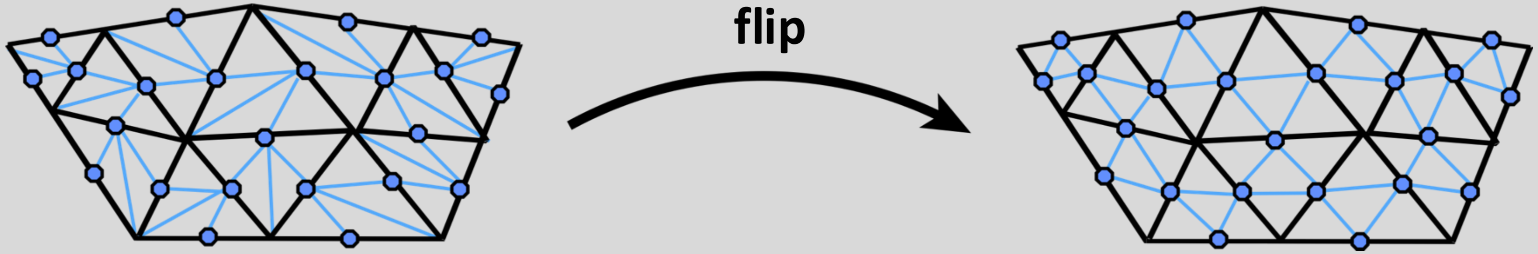


Loop Subdivision Using Local Ops

Step 1:
Split all edges in any order



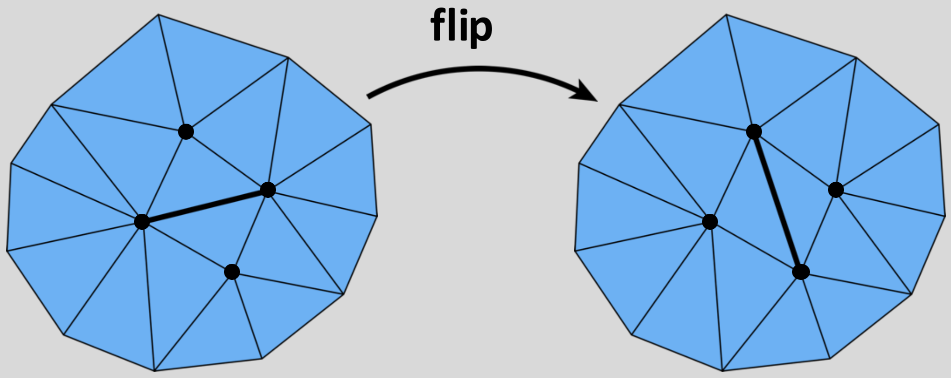
Step 2:
Flip new edges until they touch two new vertices



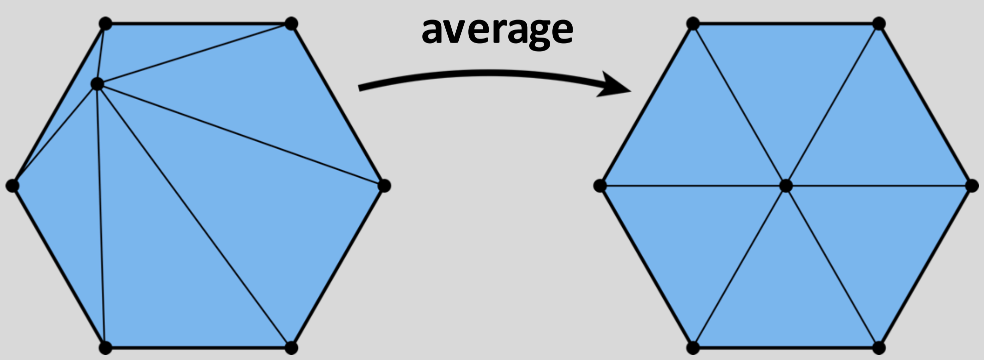
Isotropic Remeshing

Improving mesh quality (polygon angles, edge lengths, etc.) by iteratively performing local operations.

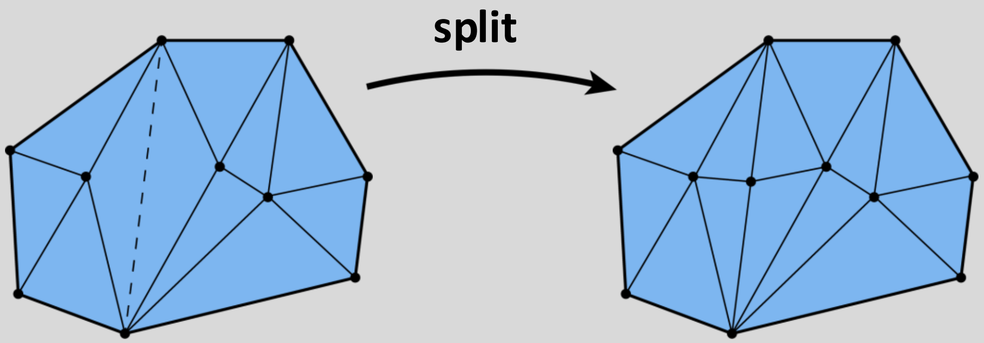
Step 1:



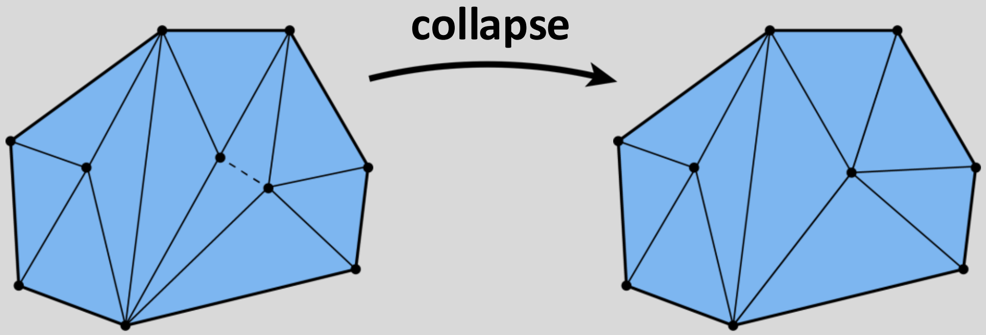
Step 2:



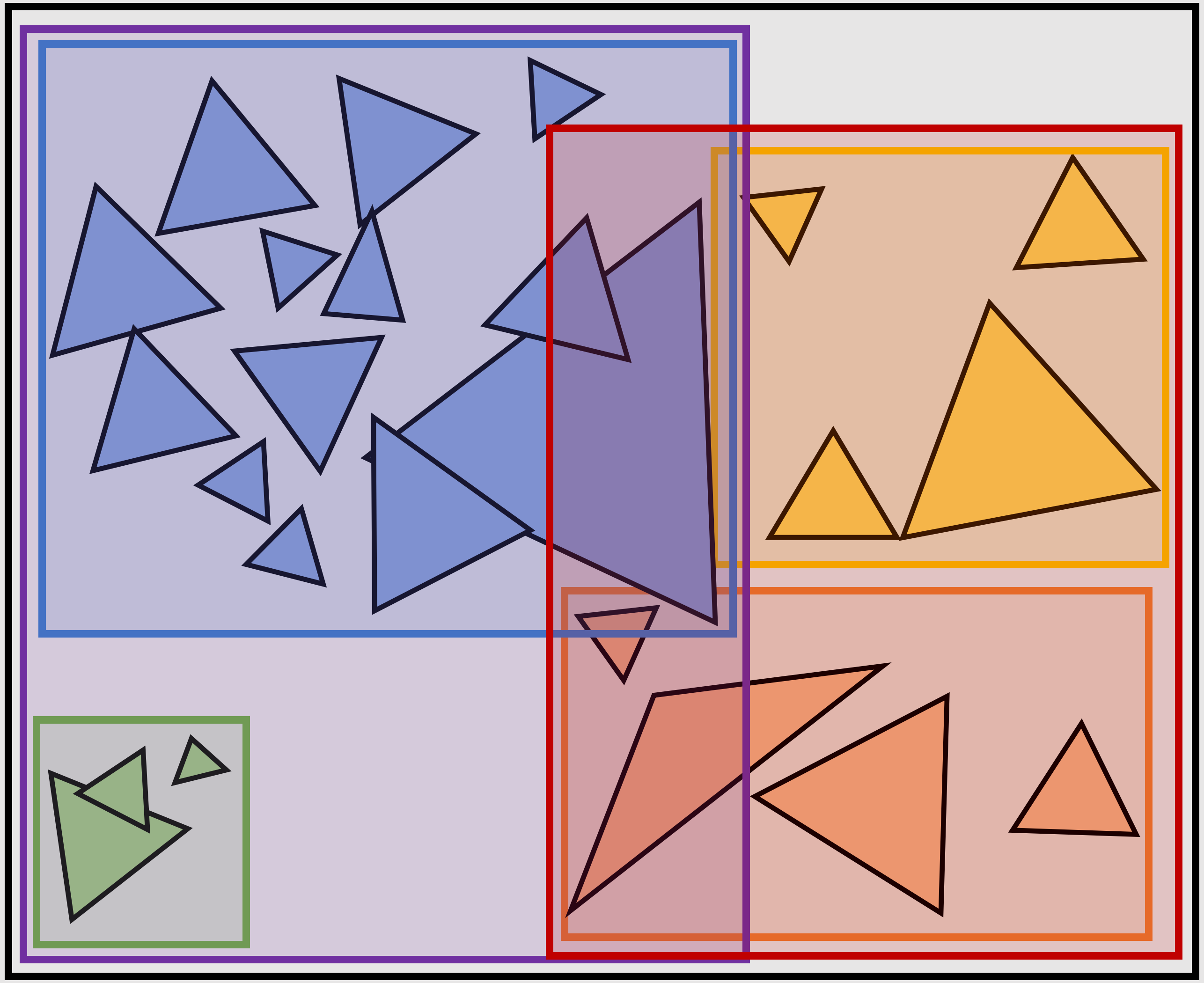
Step 3:



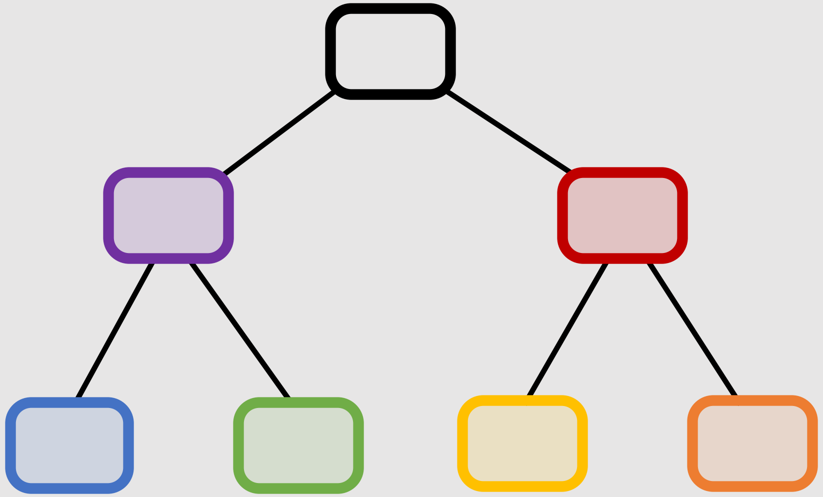
Step 4:



Bounding Volume Hierarchy (BVH)



A tree structure designed to accelerate geometry queries, e.g. ray-triangle intersections, by efficiently reducing the number of necessary checks.



Bounding boxes will sometimes intersect!

BVH Construction and Traversal

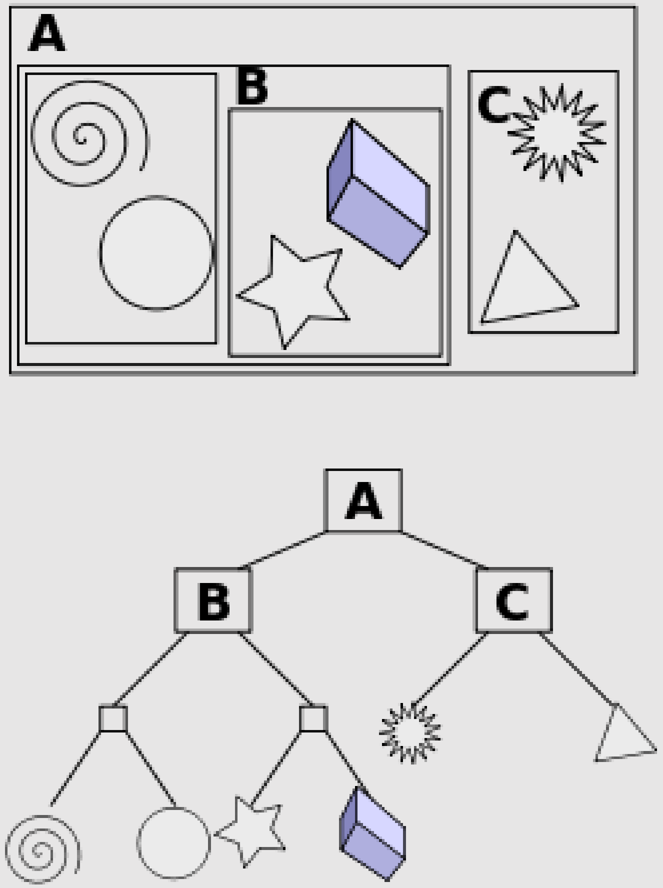
Building the BVH:

1) Pick axis [x,y,z]
    1) Sort primitives on axis by centroid
    2) Bin primitives (B = 32)
    3) Partition primitives by bin along axis
    4) Compute cost, saving best result
2) Construct 2 child nodes from best cost result
3) Recurse until few primitives (< 4) left in node

Traversing the BVH:

1) Check if ray hits current node bbox
2) If hit, find which child node is closer to ray
3) Recurse down closer child
4) If the farther child node is closer to the ray than the hit discovered, recurse down the farther child

Traversal cost is $O(\log(N))$, same as tree-search

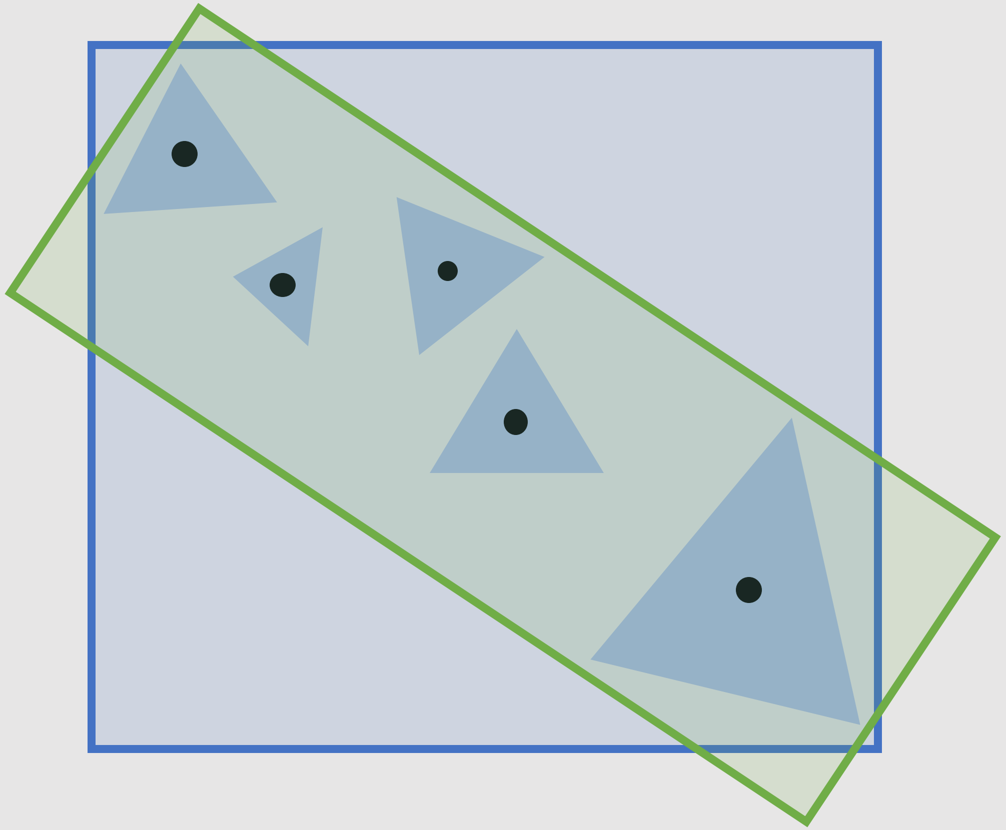


Non-Axis-Aligned BVH

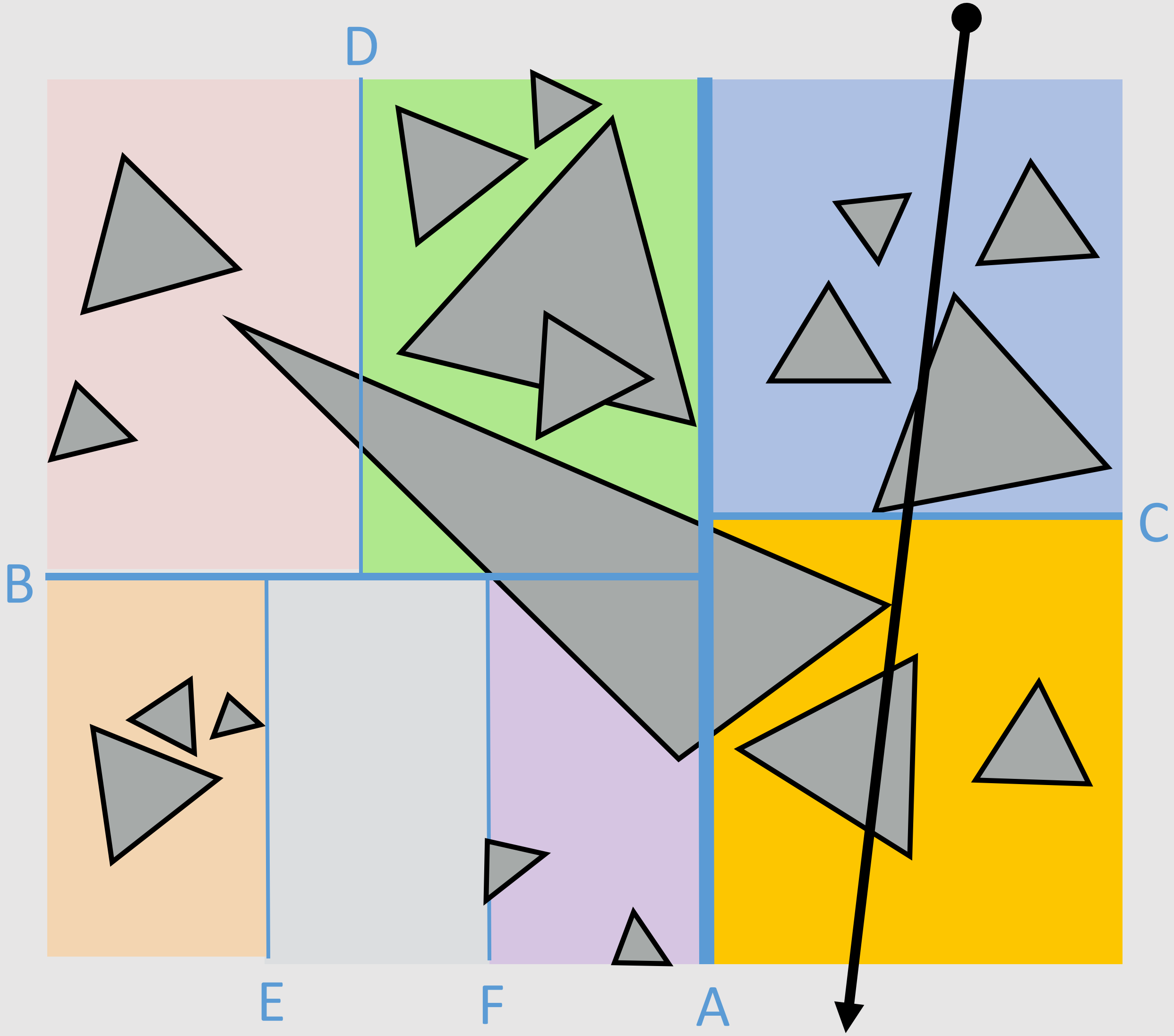
- **What is an axis-aligned BVH?**
  - By searching for partitions along the axes [x,y,z], we are constraining ourselves to build partitions with bounding boxes that are axis-aligned

- **How do we make a non-axis-aligned BVH?**
  - Simple! Just search for partitions that are not constrained to [x,y,z]
    - Easy in theory, difficult in practice

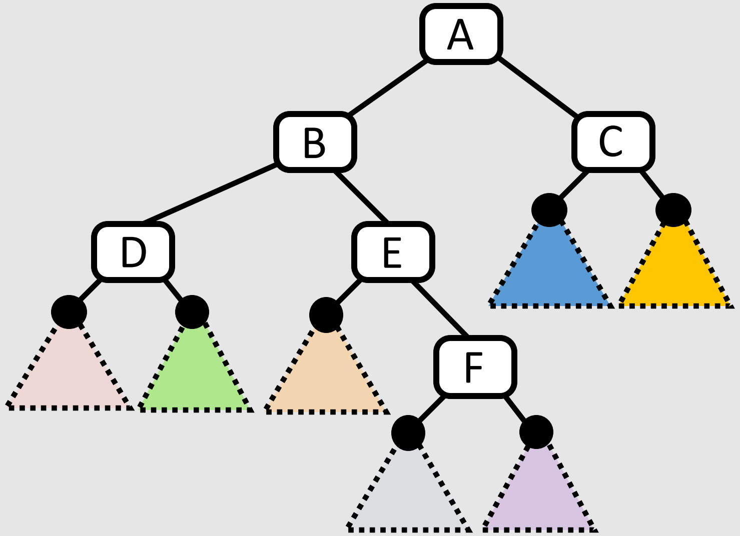
- **What are the pros/cons of non-axis-aligned BVH?**
  - [+] Better cost
  - [+] Nodes have less likelihood of having empty space
  - [-] More work to compute partitions
  - [-] Larger cost checking intersection for non-aligned bboxes
  - [-] More memory overhead



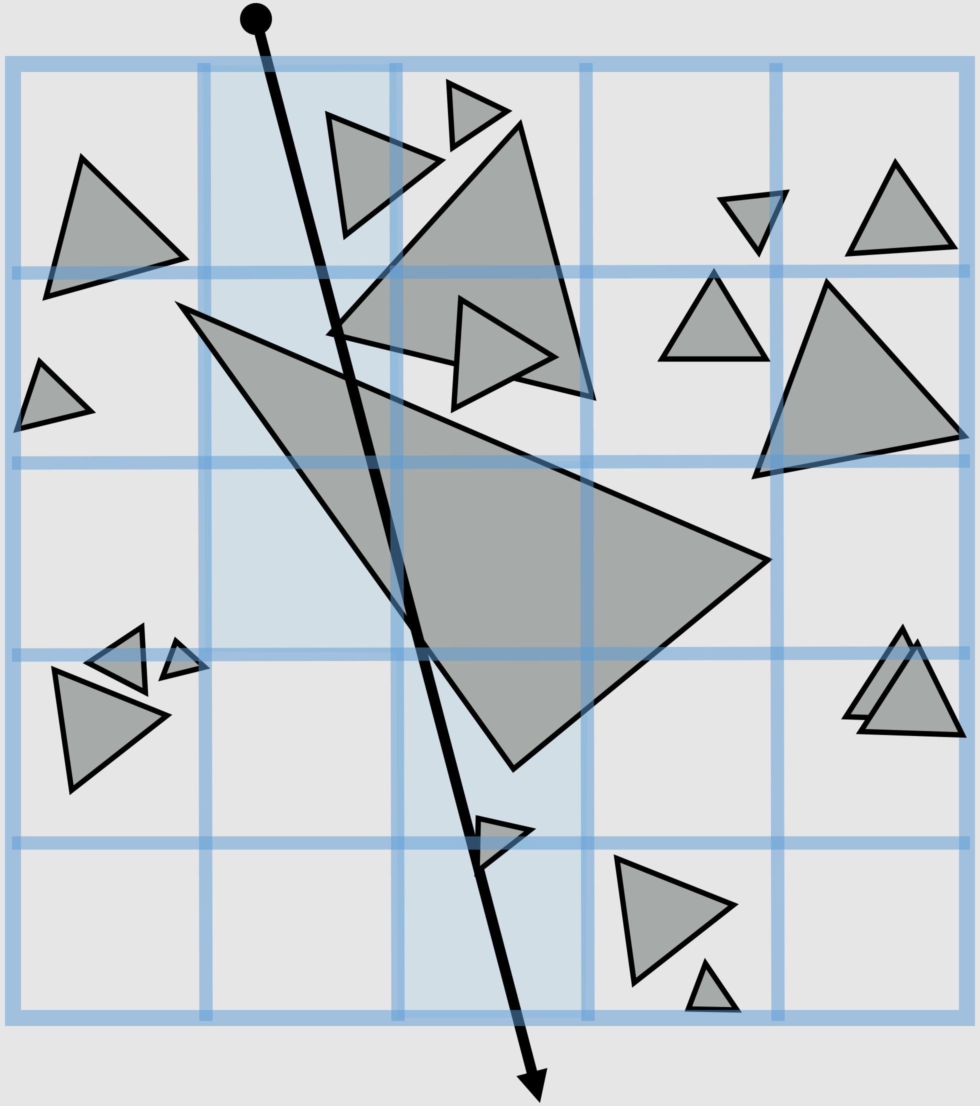
K-D Trees



- Recursively partition space via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal proceeds in front-to-back order
  - Unlike BVH, can terminate search after first hit is found
  - Still $O(\log(N))$ performance



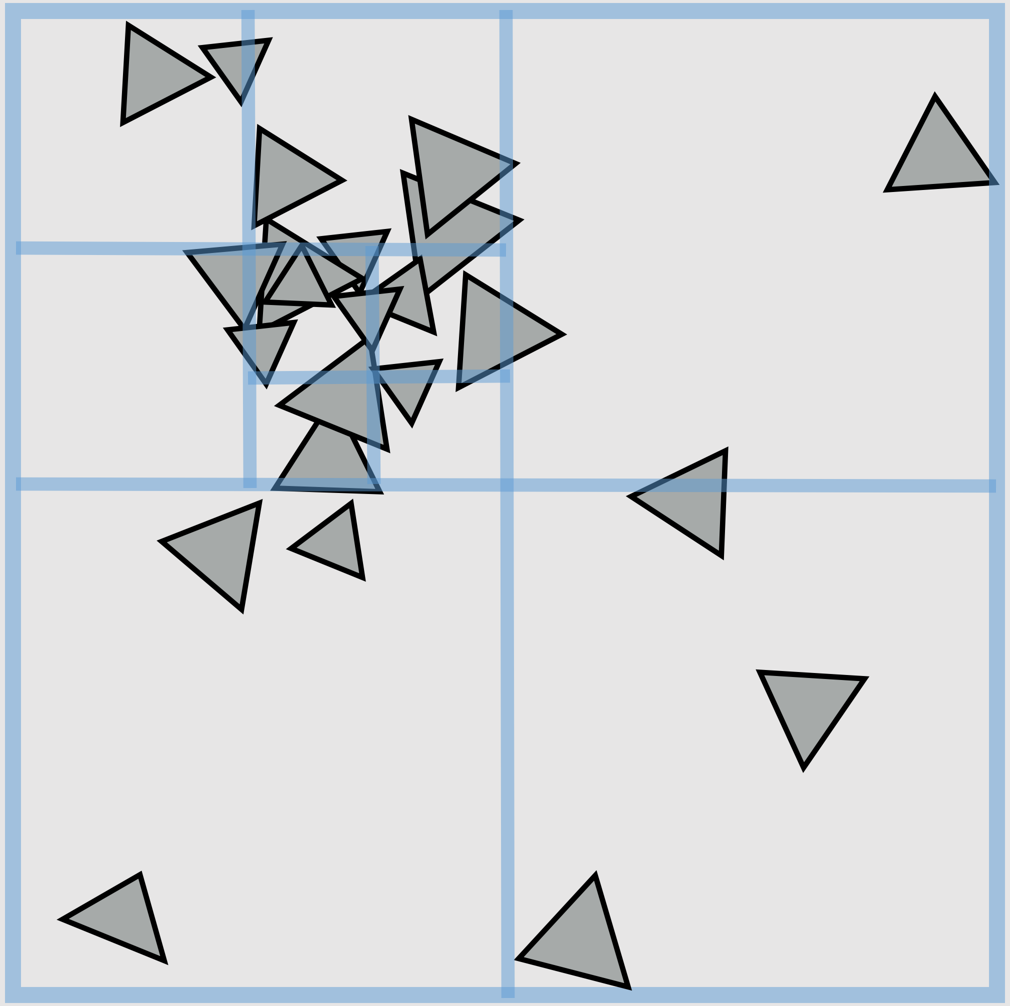
Uniform Grid



- Partition space into equal sized volumes (volume-elements or “voxels”)
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: 3D line rasterization)
  - Only consider intersection with primitives in voxels the ray intersects

- What is a good number of voxels?
  - Should be proportional to total number of primitives N
  - Number of cells traversed is proportional to $O(\sqrt[3]{N})$
    - A line going through a cube is a cubed root
    - Not as good as $O(\log(N))$

Quad-Tree/Octree



- Like uniform grid, easy to build
- Has greater ability to adapt to location of scene geometry than uniform grid
  - Still not as good adaptability as K-D tree

- **Quad-tree:** nodes have 4 children
  - Partitions 2D space
- **Octree:** nodes have 8 children
  - Partitions 3D space

- ~~A1: Rasterization~~
- ~~A2: Geometry~~
- A3: Rendering
- A4: Animation

Color & Radiometry

- Absorption vs Emission

- Eyes vs Cameras
  - Pupil
  - Lens
  - Rods
  - Cones

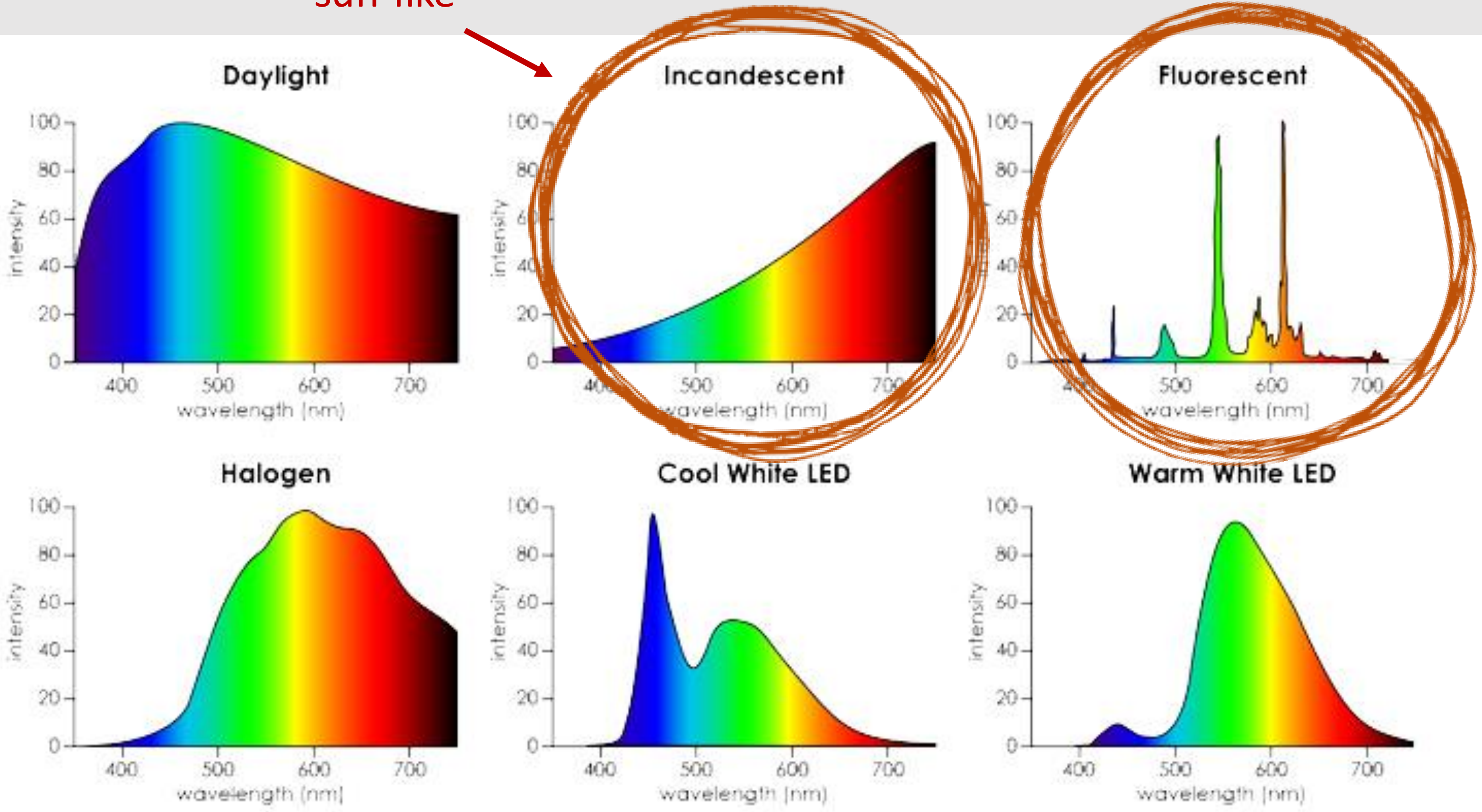
- Radiance
  - Radiant Energy
  - Radiant Energy Density
  - Radiant Flux
  - Irradiance

- Lambert’s Law

Emission Spectrum Examples

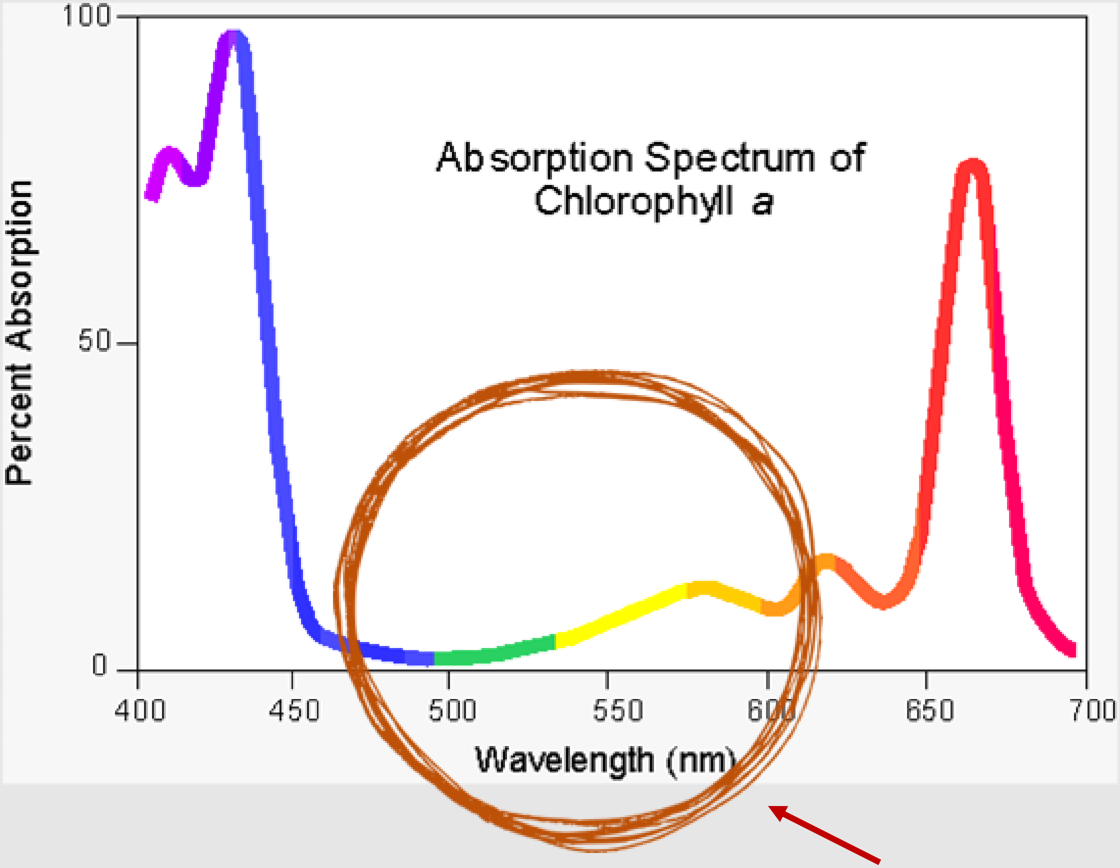
sun-like

energy efficient



admesy
ADVANCED MEASUREMENT SYSTEMS

Absorption Spectrum Examples



plants are green because they do not absorb green light

‘Eye’ See What You Mean

- Eyes are biological cameras
  - Light passes through the pupil [black dot in the eye]
  - Iris controls how much light enters eye [colored ring around pupil]
    - Eyes are sensitive to too much light
    - Iris protects the eyes
  - Lens behind the eye converges light rays to back of the eye
    - Ciliary muscles around the lens allow the lens to be bent to change focus on nearby/far objects

- 130+ million retina cells at the back of the eye
  - Cells pick up light and convert it to electrical signal
  - Electric signal passes through optic nerve to reach the brain

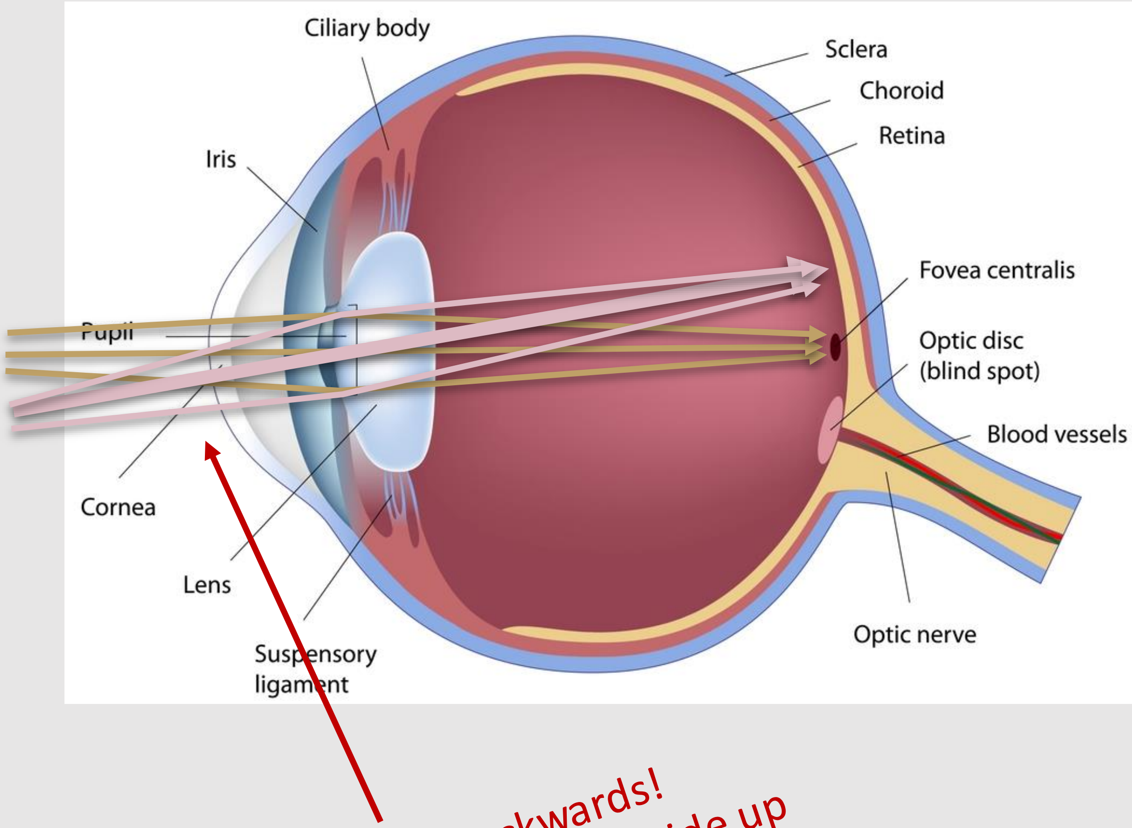
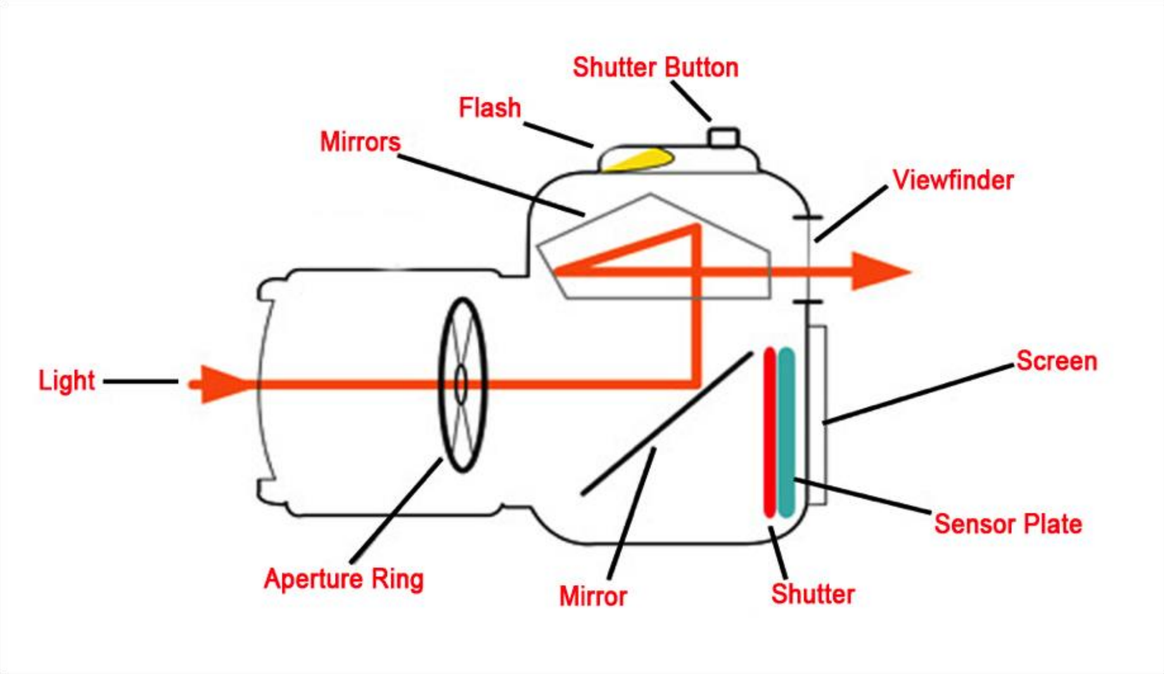


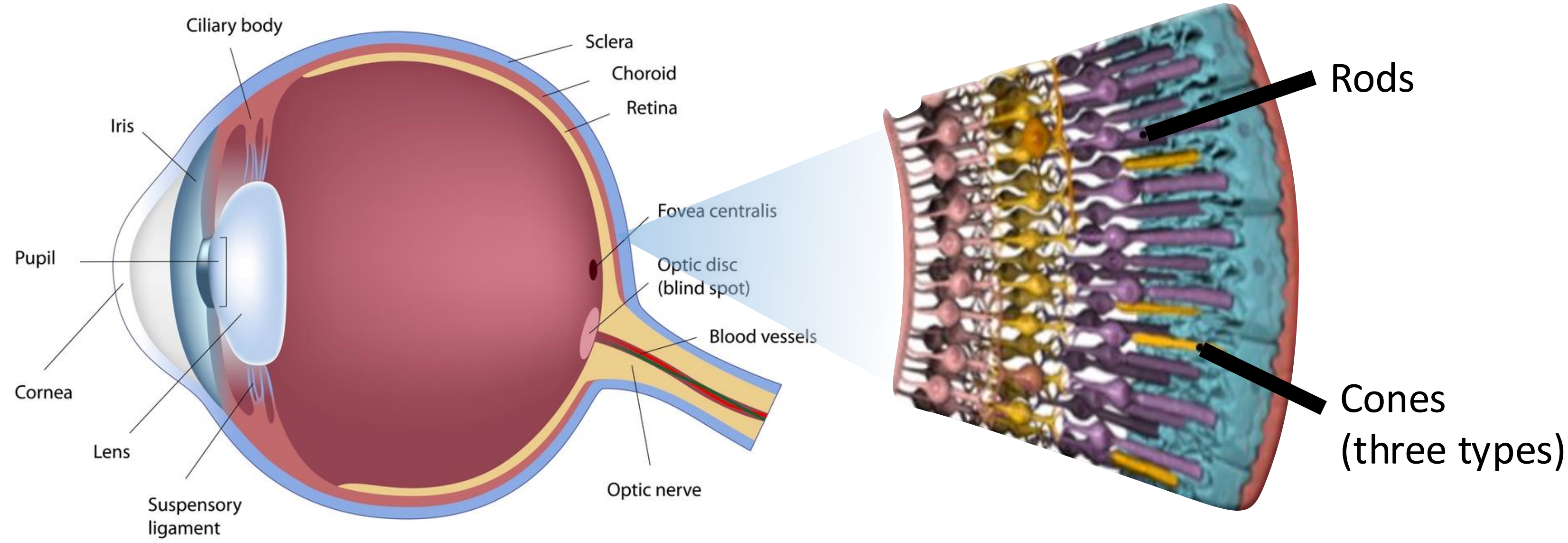
Image appears backwards!
Don’t worry, brain flips it right-side up

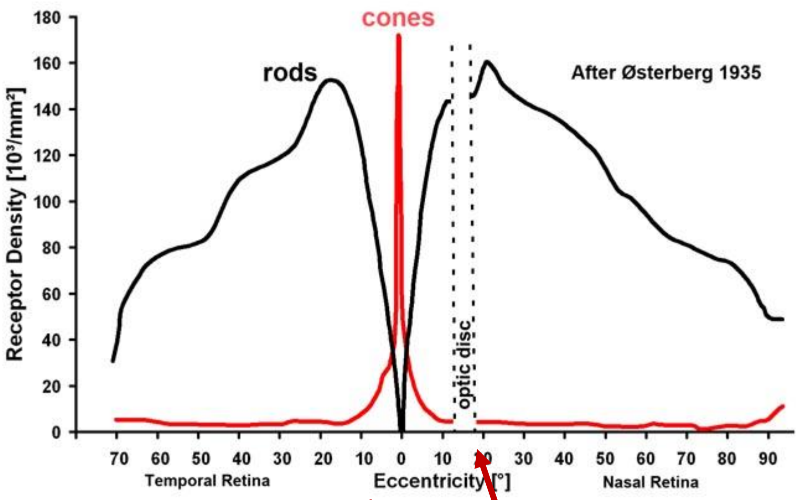
The Biological Camera



- **Pupil** is the camera opening
  - Allows light through
- **Iris** is the aperture ring
  - Controls aperture
- **Lens** is the…well, lens
  - Can change focus
- **Retina** is the sensor
  - Converts light into electrical signal
- **Brain** is the CPU
  - Performs additional compute to correct raw image signal

Rods & Cones





Best vision at center of cones!

Human blind spot

- Cones are primary receptors near fovea used under high-light viewing conditions
  - Approx. 6-7 million cones in the human eye
  - Capture color
- Rods are primary receptors far from fovea used under low-light viewing conditions
  - Approx. 120 million rods in human eye
  - Capture intensity

Spectral Response of Cones

- Long, Medium, and Small cones pick up Long, Medium, and Small wavelengths respectively
- Each cone picks up a range of colors given by their response functions
  - Not much different than absorption spectrum

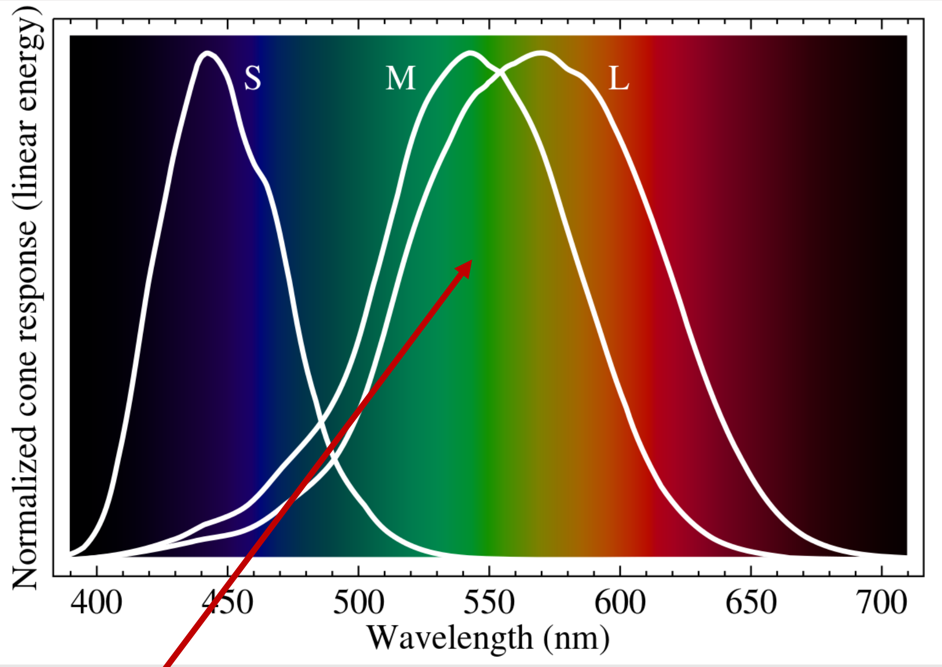
- Each cone integrates the emission & response to produce a single signal to transmit to the brain

$$S = \int_\lambda \Phi(\lambda) S(\lambda) d\lambda$$

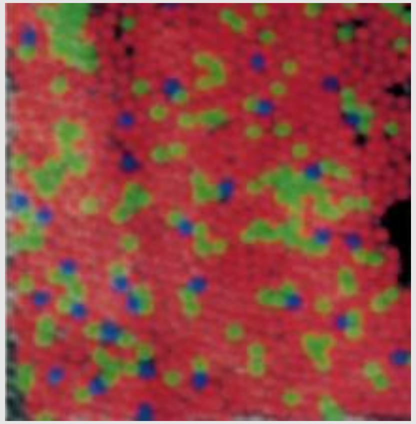
$$M = \int_\lambda \Phi(\lambda) M(\lambda) d\lambda$$

$$L = \int_\lambda \Phi(\lambda) L(\lambda) d\lambda$$

- Uneven distribution of cone types in eye
  - ~64% L cones, ~ 32% M cones ~4% S cones



A lot of green picked up!



Radiant Recap

Radiant Energy
(total number of hits)
Joules (J)

Radiant Energy Density
(hits per unit area)
Joules per sq meter (J/m^2)

Radiant Flux
(total hits per second)
Watts (W)

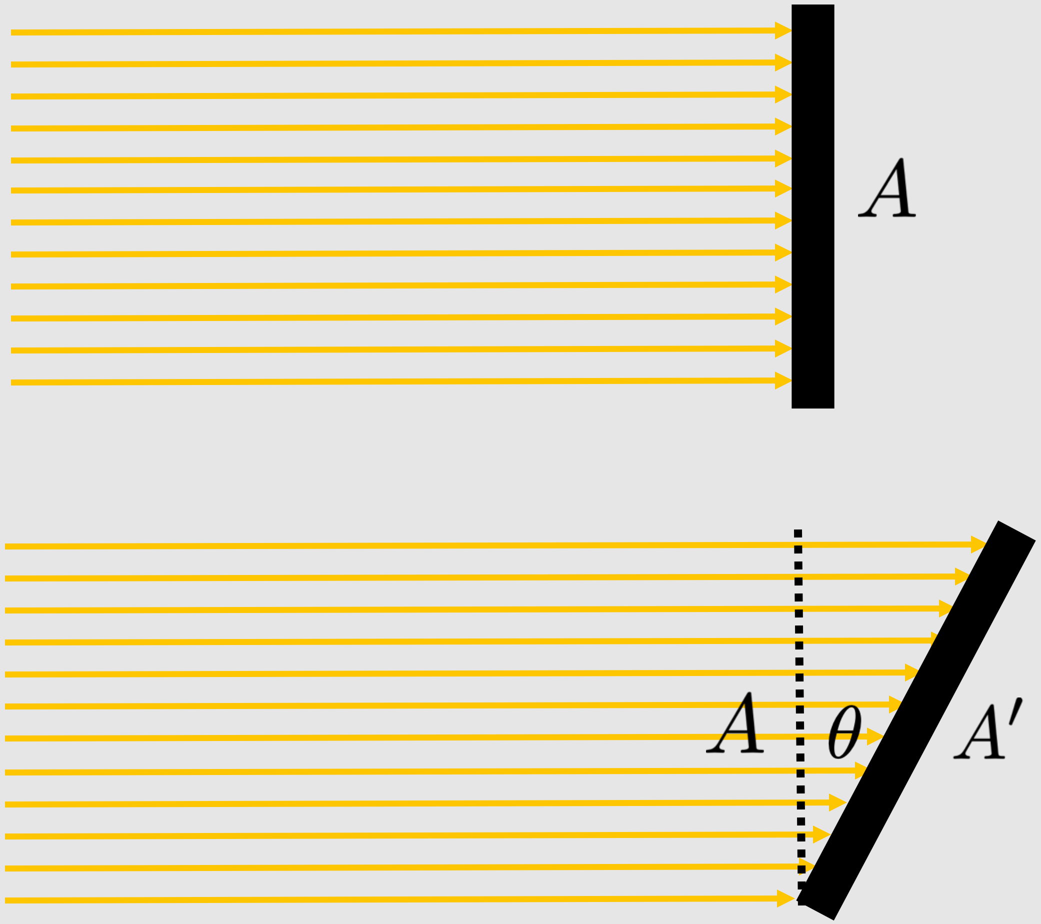
Radiant Flux Density
a.k.a. *Irradiance*
(hits per second per unit area)
Watts per sq meter(W/m^2)

Lambert’s Law

- Irradiance (E) at surface is proportional to the flux (Φ) and the cosine of angle (θ) between light direction and surface normal:

$$E = \frac{\Phi}{A'} = \frac{\Phi \cos\theta}{A}$$

- Consider rotating a plane away from light rays
  - Plane will darken until it is perpendicular to light rays, then it will be completely black



$$A = A' \cos\theta$$

The Rendering Equation

- The Rendering Equation

- Rendering Methods
    - Forwards Path-Tracing
    - Backwards Path-Tracing
    - Bi-Directional Path-Tracing
    - Metropolis Light Transport

- Variance Reduction
    - Sampling Rate
    - Ray Depth

- BRDFs
    - Lambertian
    - Mirror
    - Glass

The Rendering Equation

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o)+\int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i\to\omega_o)L_i(\mathbf{p},\omega_i)\cos\theta\,d\omega_i$$

$L_o(\mathbf{p},\omega_o)$	outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o
$L_e(\mathbf{p},\omega_o)$	emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o
$f_r(\mathbf{p},\omega_i\to\omega_o)$	scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o
$L_i(\mathbf{p},\omega_i)$	incoming radiance to point $\mathbf{p}$ from direction ω_i

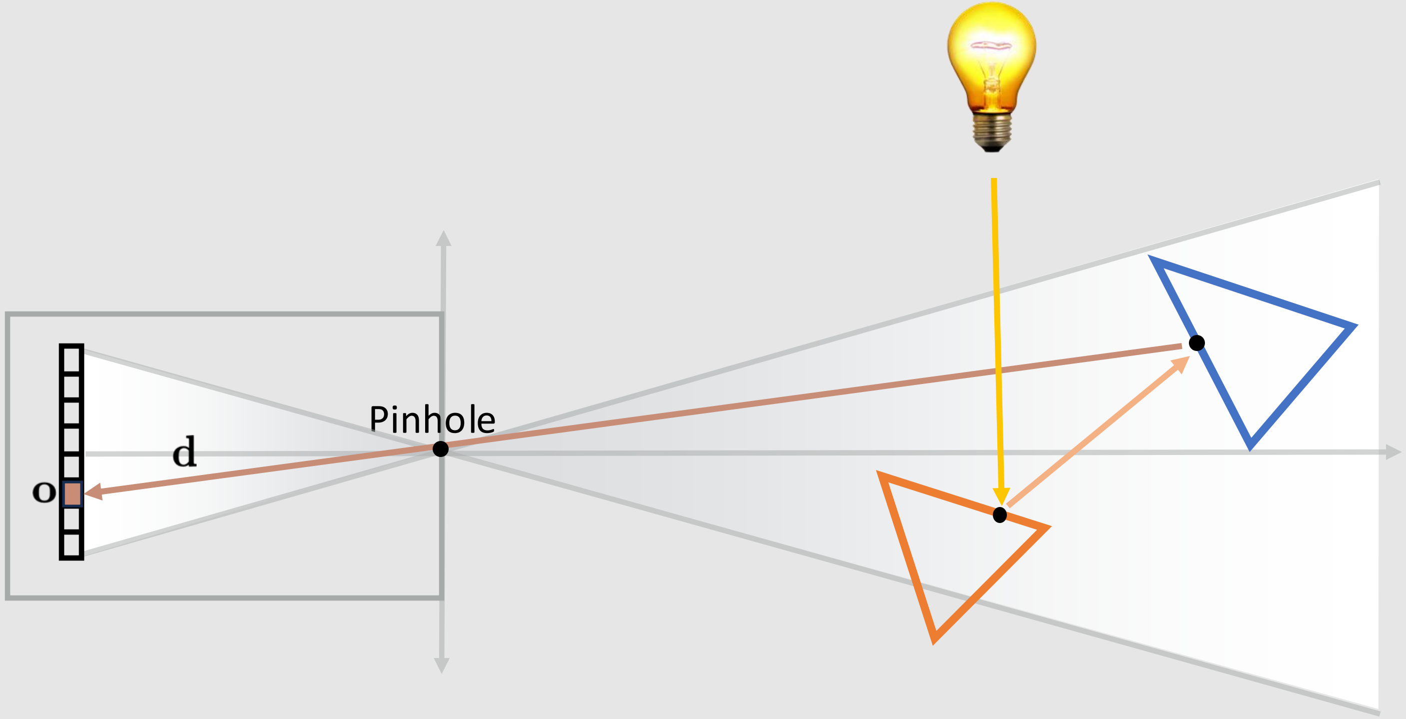
Example Of A Simple Renderer

- Yellow light ray generated from light source

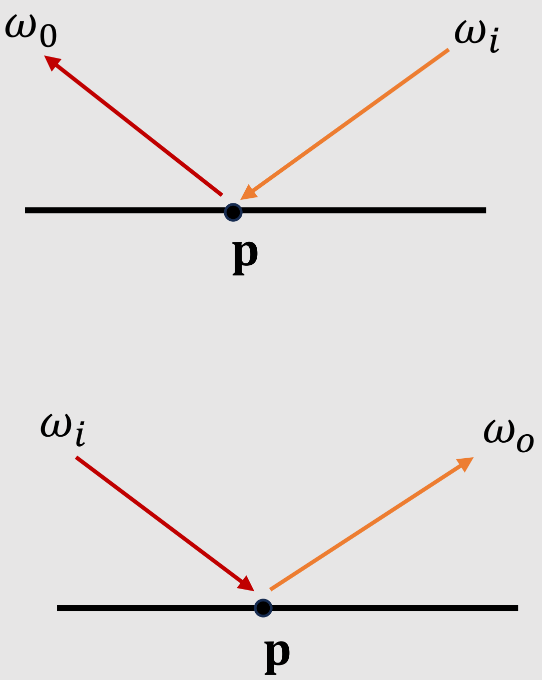
- Ray hits orange specular surface
  - Emits a ray in reflected direction
  - Mixes yellow and orange color

- Ray hits blue specular surface
  - Emits a ray in reflected direction
  - Mixes blue and yellow and orange

- Ray passes through pinhole camera
  - Light recorded on photoelectric cell
  - Incident pixel will be brown in final image



Hemholtz Reciprocity



- Reversing the order of incoming and outgoing light does not affect the BRDF evaluation
    - $f_r(\mathbf{p}, \omega_i \rightarrow \omega_o) = f_r(\mathbf{p}, \omega_o \rightarrow \omega_i)$

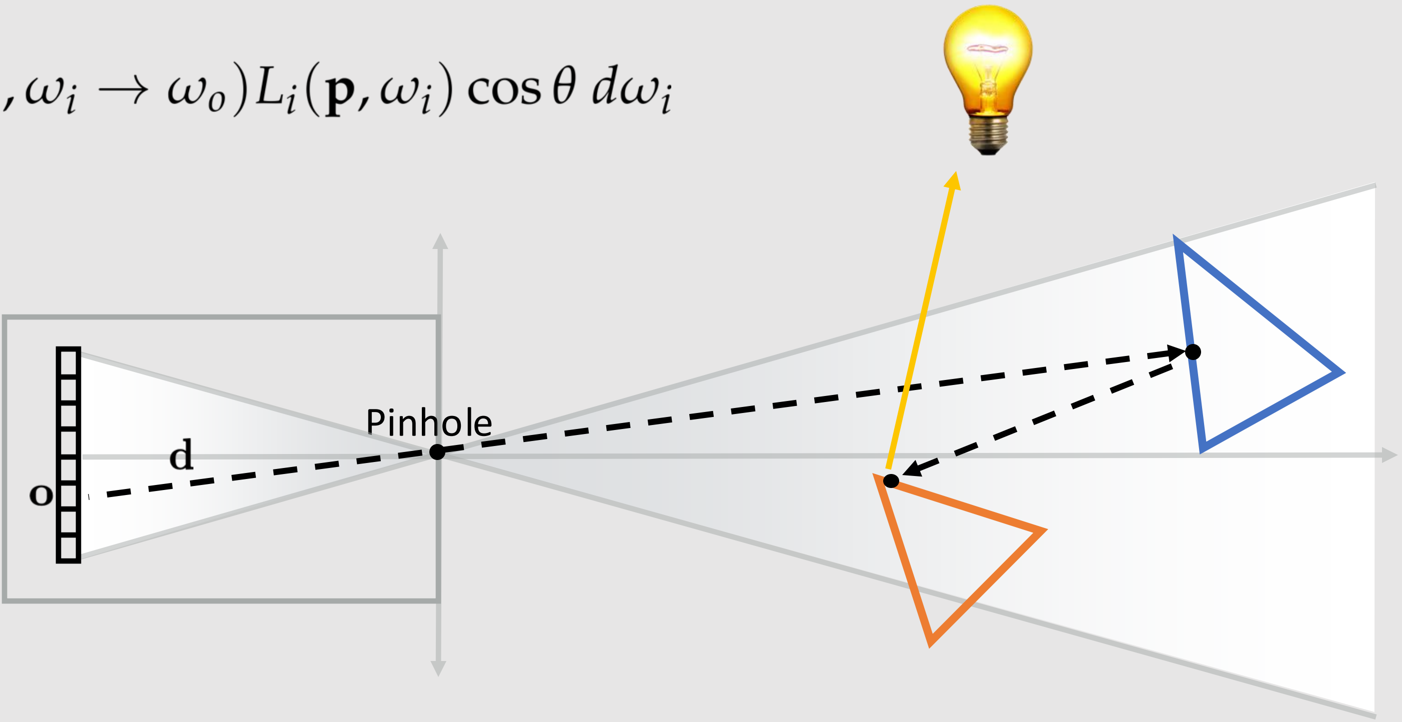
- Critical to reverse pathtracing algorithms
    - Allows us to trace rays backwards and still get the same BRDF affect

Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \rightarrow \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

- Intersect △ , no emission ☐
- Intersect △ , no emission ☐
- Ray terminate, emission 🟨



$$L(pixel) = L_e(ray_1) + f_r(obj_1)[L_e(ray_2) + f_r(obj_2)[L_e(ray_3)]]$$

$$L(pixel) = \square + f_r(\triangle)[\ \square + f_r(\triangle)[\ \blacksquare\ ]]$$

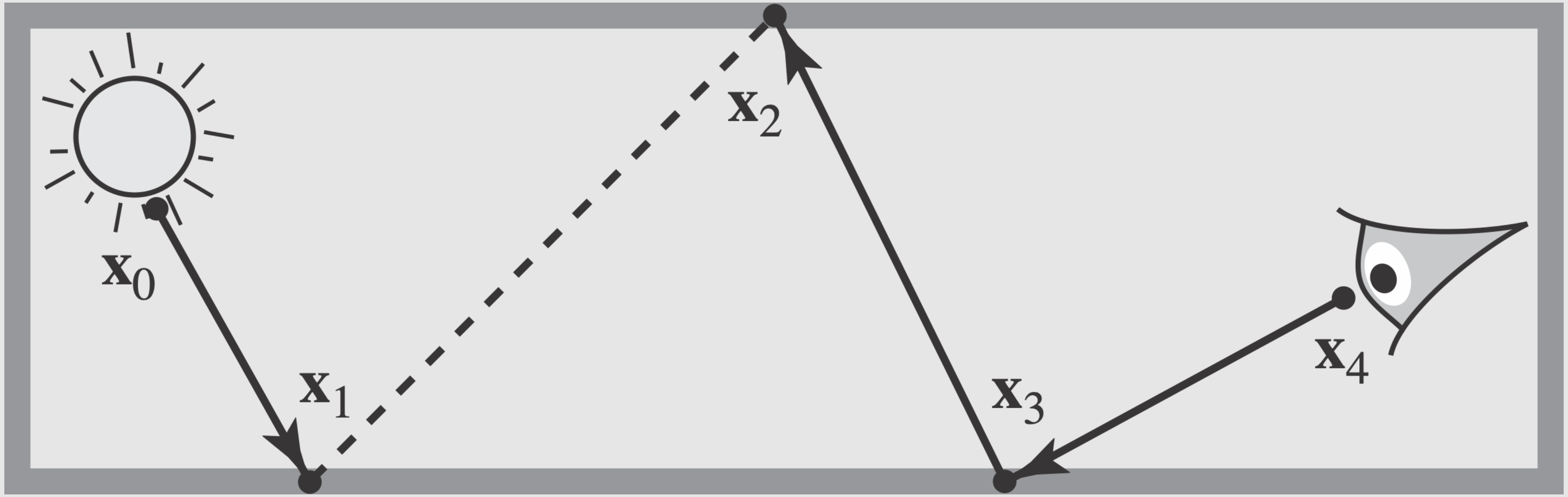
Bidirectional Path Tracing

- If path tracing is so great, why not do it **twice**?
    - Main idea of bidirectional!

- Trace a ray from the camera into the scene
- Trace a ray from the light into the scene
    - Connect the rays at the end

- Unbiased algorithm
    - No longer trying to connect rays through non-volume sources

- Can set different lengths per ray
    - Example: Forward m = 2, Backward m = 1

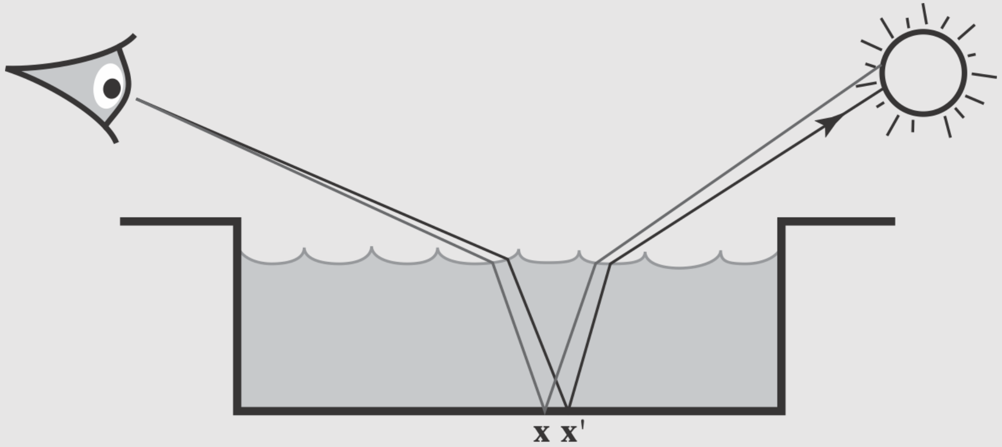


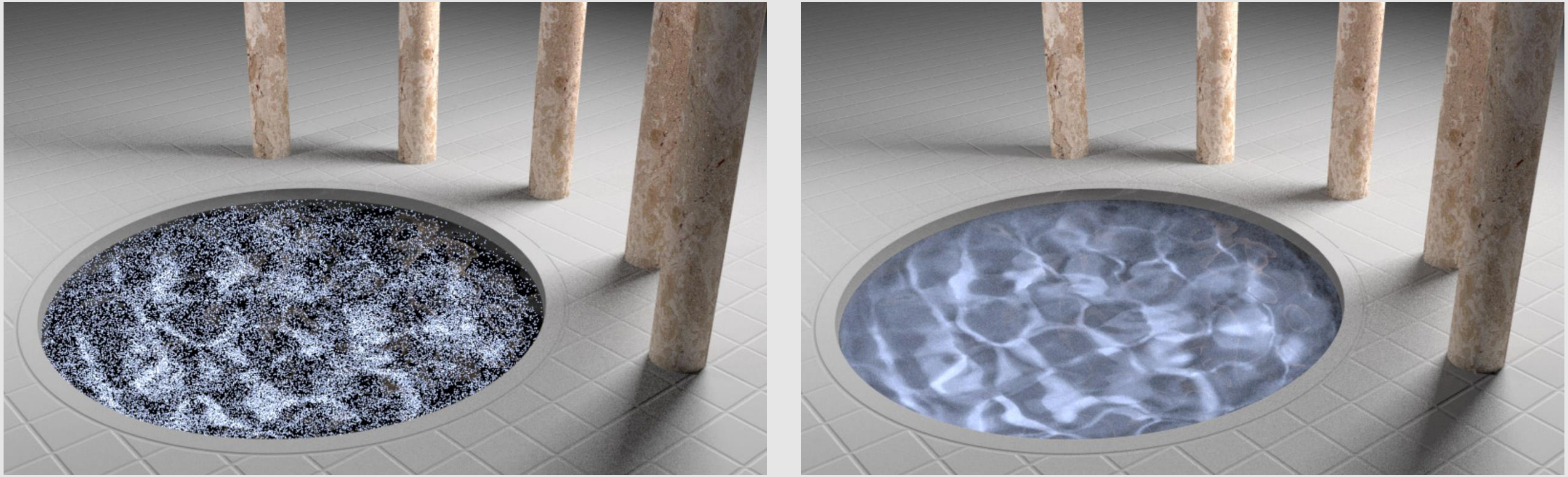
Metropolis Light Transport

- **Similar idea:** mutate good paths

- Water causes paths to refract a lot
    - Small mutations allows renderer to find contributions faster

- Path Tracing and MLT rendered in the same time





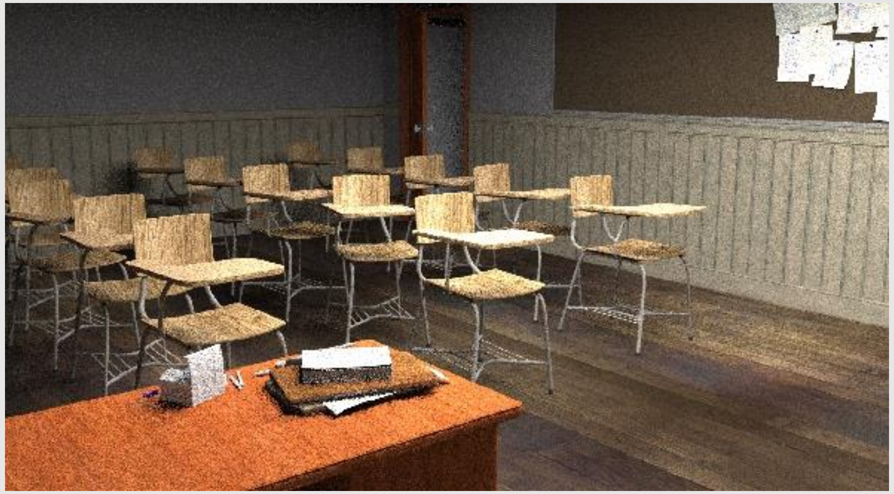
[Path Tracing] **[Metropolis Light Transport]**

Number Of Ray Samples

- **Number of Rays**
    - How many rays we trace into the scene
        - Measured as samples (rays) per pixel [spp]

- Increasing the number of rays increases the quality of the image
    - Anti-aliasing
    - Reduces black spots from terminating emission occlusion

[1 spp]



[16 spp]

Number Of Ray Bounces

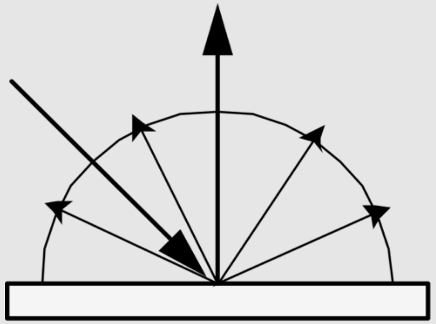
- **Number of Ray Bounces**
    - How many times a ray bounces before it terminates
        - Measured as ray bounce or depth

- Increasing the number of ray bounces increases the quality of the image
    - Better color blending around images
    - More details reflected in specular images

[2 depth]

[8 depth]

Lambertian Material



- Also known as diffuse
- Light is equally likely to be reflected in each output direction
  - BRDF is a constant, relying on albedo (ρ)

$$f_r = \frac{\rho}{\pi}$$

  - BRDF can be pulled out of the integral

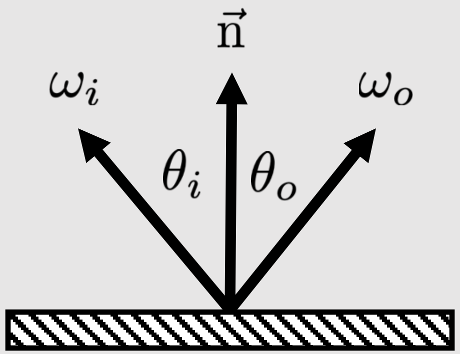
$$\begin{aligned} L_o(\omega_o) &= \int_{H^2} f_r\, L_i(\omega_i)\ \cos\theta_i\, \mathrm{d}\omega_i \\ &= f_r \int_{H^2} L_i(\omega_i)\ \cos\theta_i\, \mathrm{d}\omega_i \\ &= f_r E \end{aligned}$$

- Easy! Pick any outgoing ray w_o

Minions (2015) Illumination Entertainment

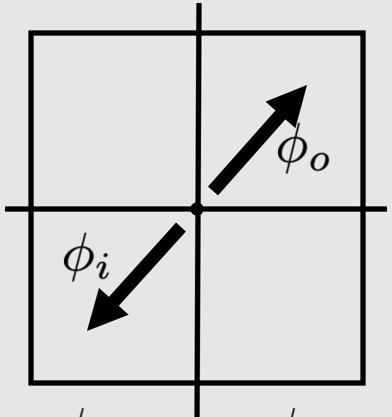
Reflective Material

[side view]



$$\theta = \theta_o = \theta_i$$

[top view]



$$\phi_o = -\phi_i$$

- Reflectance equation described as:

$$\omega_o = -\omega_i + 2(\omega_i \cdot \vec{n})\vec{n}$$

  - Why is the ray ω_i pointing away from the surface?
    - Just syntax. Incoming and outgoing rays share same origin point **p**

- BRDF represented by dirac delta (δ) function:

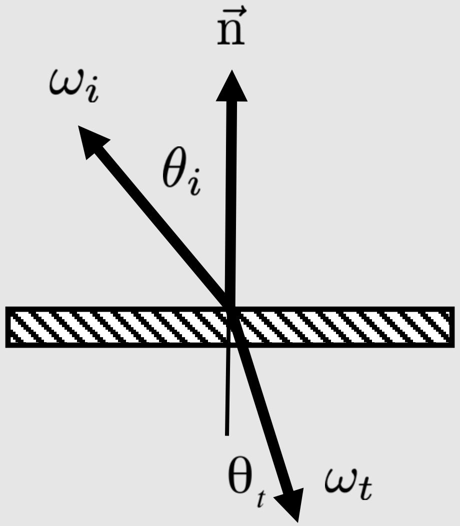
$$f_r(\theta_i, \phi_i; \theta_o, \phi_o) = \frac{\delta(\cos\theta_i - \cos\theta_o)}{\cos\theta_i}\delta(\phi_i - \phi_o \pm \pi)$$

  - 1 when ray is perfect reflection, 0 everywhere else
  - All radiance gets reflected, nothing absorbed

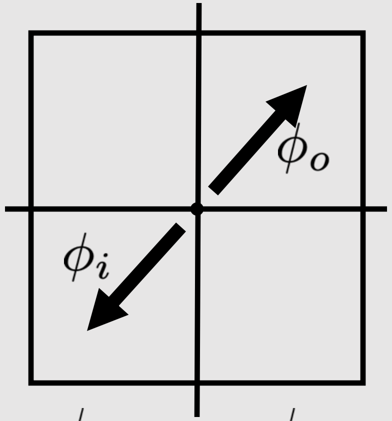
- In practice, no hope of finding reflected direction via random sampling
  - Simply pick the reflected direction!

Refractive Material

[side view]



[top view]



$$\phi_o = -\phi_i$$

- Refractive equation described as:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

  - Also known as Snell’s Law

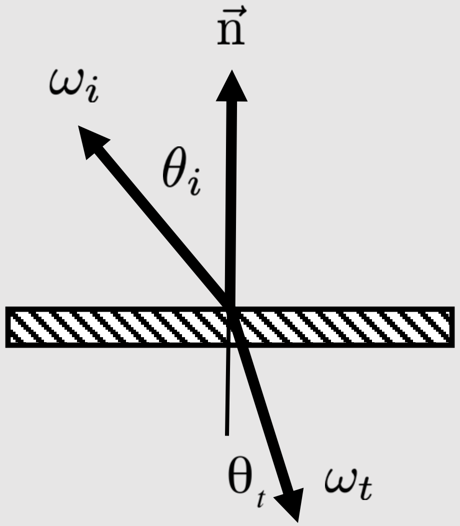
- η_i and η_t describe the index of refraction of the incoming and outgoing mediums
  - Example: η_i is air, η_t is water

Medium	$\boldsymbol{\eta}$
Vacuum	1.0
Air (sea level)	1.00029
Water (20°C)	1.333
Glass	1.5-1.6
Diamond	2.42

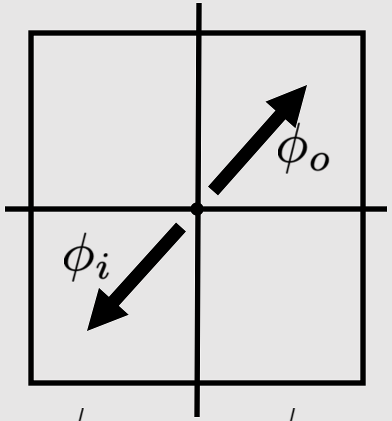
  - η is the ratio of the speed of light in a vacuum to that in a second medium of greater density
    - The larger the η, the denser the material

Refractive Material

[side view]



[top view]



$$\phi_o = -\phi_i$$

- Refractive equation described as:

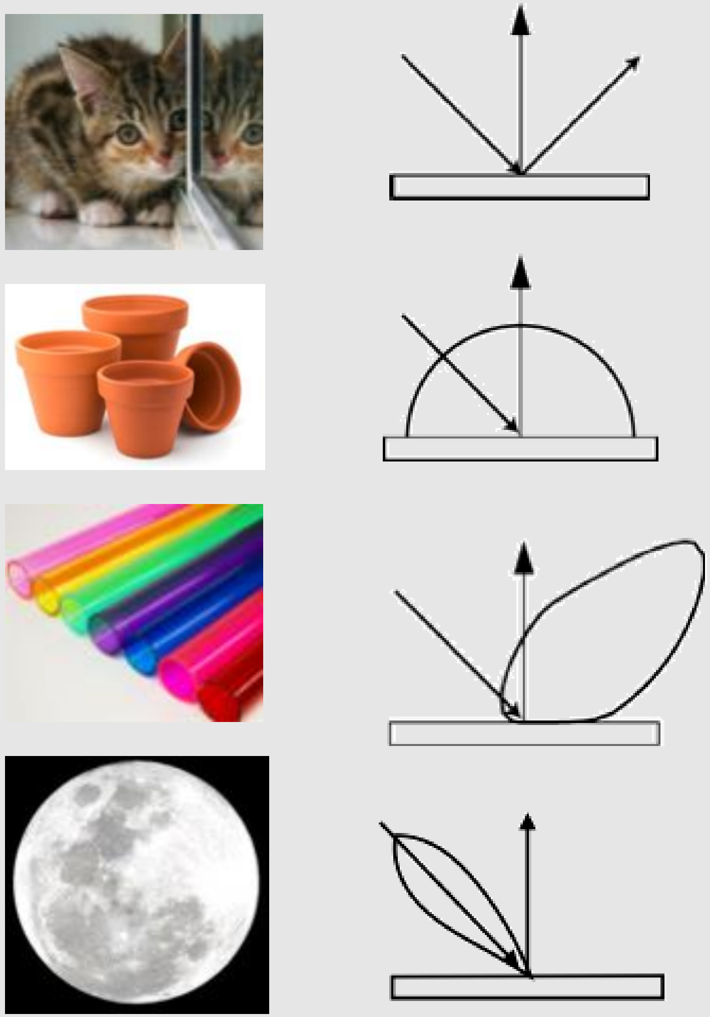
$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

  - Also known as Snell's Law

- Can rewrite the equation as:

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t}$$
$$= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 \sin^2\theta_i}$$
$$= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2\ \theta_i)}$$

Types of Reflectance Functions



Ideal Specular

- Perfect mirror

Ideal Diffuse

- Uniform in all directions

Glossy Specular

- Majority of light in reflected direction

Retroreflective

- Reflects light back towards source

- ~~A1: Rasterization~~
- ~~A2: Geometry~~
- ~~A3: Rendering~~
- A4: Animation

Animation Simulation

- Simulation
    - ODEs vs PDEs
    - Boundary Conditions
    - Laplacian

- Motion Graphs
    - Displacement
    - Velocity
    - Acceleration

- Splines
    - Natural Splines
    - Hermite/Bezier Curves
    - B-Splines

Natural Splines

- Can build a spline out of piecewise cubic polynomials p_i
  - Each polynomial extends from range $t = [0,1]$
  - Polynomials should connect on boundary
    - Keyframes agree at endpoints [C0 continuity]:

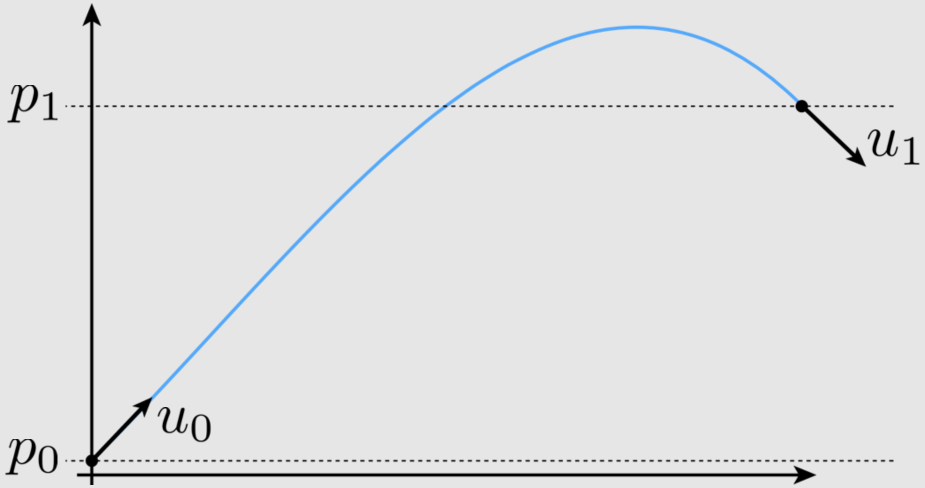
$$p_i(t_i) = f_i, \qquad p_i(t_{i+1}) = f_{i+1}, \qquad \forall i = 0, \dots, n-1$$

    - Tangents agree at endpoints [C1 continuity]:

$$p'_i(t_{i+1}) = p'_{i+1}(t_{i+1}), \qquad \forall i = 0, \dots, n-2$$

    - Curvature agrees at endpoints [C2 continuity]:

$$p''_i(t_{i+1}) = p''_{i+1}(t_{i+1}), \qquad \forall i = 0, \dots, n-2$$



- Total equations:
  - 2n + (n-1) + (n-1) = 4n – 2
- Total DOFs:
  - 2n + n + n = 4n
- Set curvature at endpoints to 0 and solve

$$p''_0(t_0) = 0, \qquad p''_0(t_{i+1}) = 0$$

Hermite/Bézier Splines

- Each cubic "piece" specified by endpoints and tangents
  - Keyframes set at endpoints:

$$p_i(t_i) = f_i, \qquad p_i(t_{i+1}) = f_{i+1}, \qquad \forall i = 0, \ldots, n-1$$

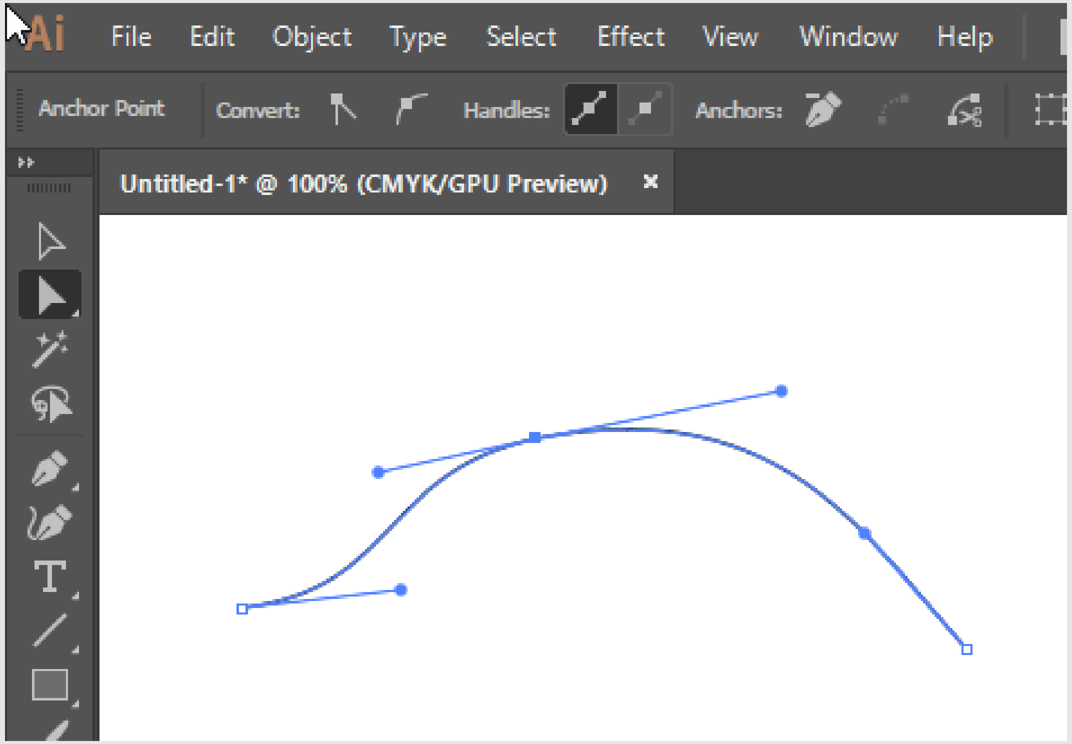
  - Tangents set at endpoint:

$$p'_i(t_i) = u_{i,} \qquad p'_i(t_{i+1}) = u_{i,+1}, \qquad \forall i = 0, \ldots, n-1$$

- Natural splines specify just keyframes
  - Bezier splines specify keyframes and tangents
  - Can get continuity if tangents are set equal

- Total equations:
  - 2n + 2n = 4n

- Commonly used in vector art programs
  - Illustrator
  - Inkscape
  - SVGs



B-Splines

- Compute a weighted average of nearby keyframes when interpolating
- B-spline basis defined recursively, with base condition:

$$B_{i,1}(t) := \begin{cases} 1, & \text{if } t_i \le t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

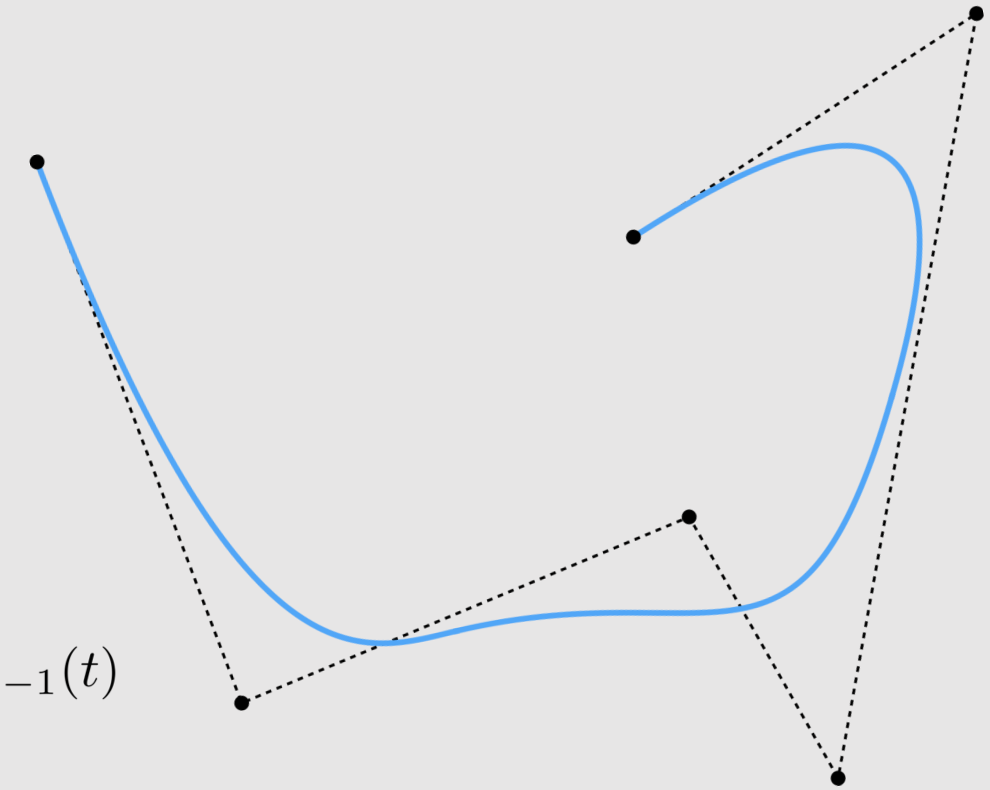
- And inductive condition:

$$B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} B_{i+1,k-1}(t)$$

- B-spline is a linear combination of bases:

$$f(t) := \textstyle\sum_i a_i B_{i,\underset{\text{degree}}{d}}$$

degree



Splines Review

	[Interpolation]	**[Continuity]**	**[Locality]**
Linear	✓	✗	✓
Natural	✓	✓	✗
Hermite	✓	✗	✓
Bezier	✓	✗	✓
Catmull-Rom	✓	✗	✓
B-Spline	✗	✓	✓

Simulations

- ODE vs PDE

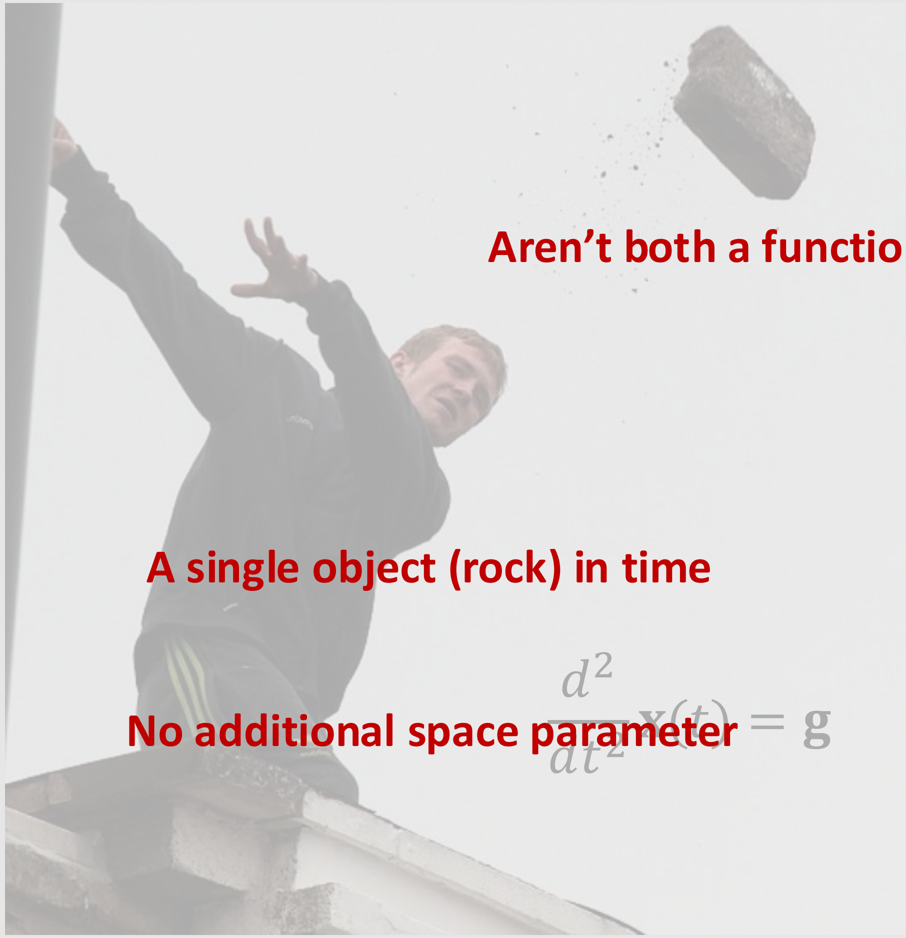
- Time Integration
    - Forward Euler
    - Symplectic Euler

- Laplacian
    - 2nd-order Derivative

- Boundary Conditions
    - Dirichlet
    - Neumann

ODEs vs. PDEs

Aren’t both a function of space and time?

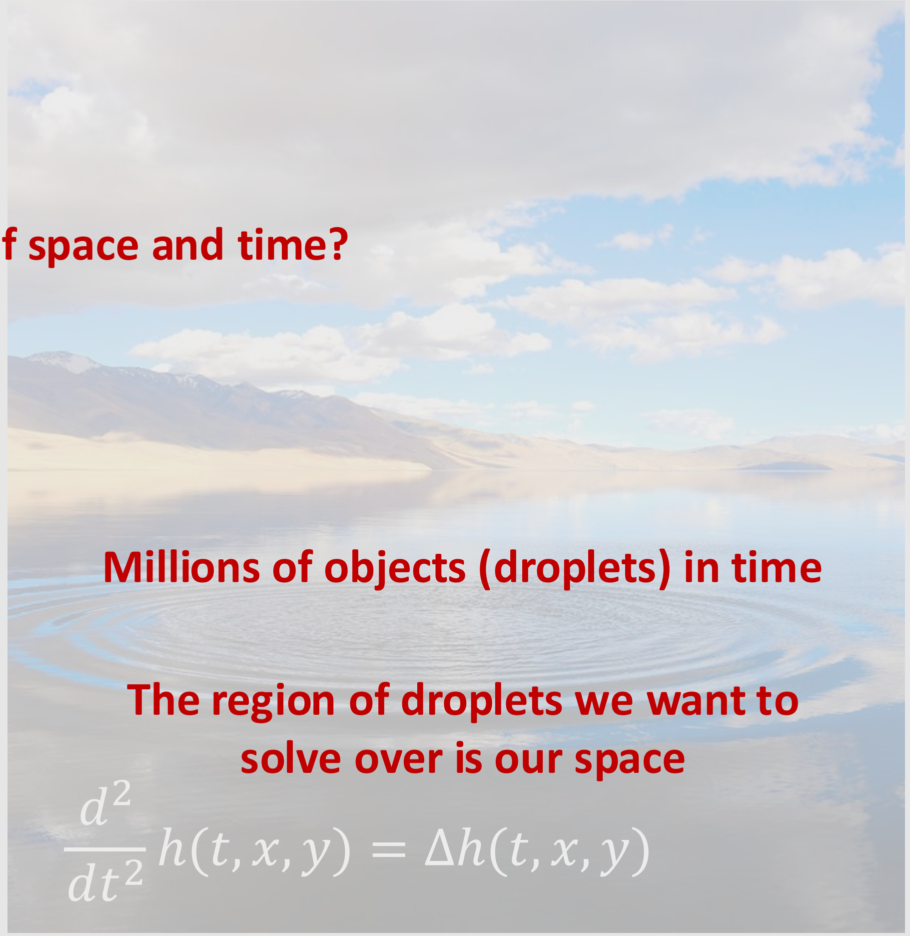


A single object (rock) in time

No additional space parameter

$$\frac{d^2}{dt^2}\mathbf{x}(t) = \mathbf{g}$$

[ODE] throwing a rock



Millions of objects (droplets) in time

The region of droplets we want to solve over is our space

$$\frac{d^2}{dt^2}h(t,x,y) = \Delta h(t,x,y)$$

[PDE] thrown rock lands in pond

Explicit Time Integration Methods

[Forward]

$$v_{k+1} = v_k + \tau * a(q_k)$$
$$q_{k+1} = q_k + \tau * v_k$$

[Symplectic]

$$v_{k+1} = v_k + \tau * a(q_k)$$
$$q_{k+1} = q_k + \tau * v_{k+1}$$

[Verlet]

$$v_{k+1} = v_{k+0.5} + \frac{\tau}{2} * a(q_k)$$
$$q_{k+1} = q_k + \tau * v_{k+1}$$
$$v_{k+1.5} = v_{k+1} + \frac{\tau}{2} * a(q_k)$$

[RK2]

$$v'_{k+1} = \tau * a(q_k)$$
$$v''_{k+1} = \tau * a(q_k + \frac{v'_{k+1}}{2})$$
$$v_{k+1} = v_k + v''_{k+1}$$
$$q_{k+1} = q_k + \tau * v_{k+1}$$

[RK4]

$$v'_{k+1} = \tau * a(q_k)$$
$$v''_{k+1} = \tau * a(q_k + \frac{v'_{k+1}}{2})$$
$$v'''_{k+1} = \tau * a(q_k + \frac{v''_{k+1}}{2})$$
$$v''''_{k+1} = \tau * a(q_k + v'''_{k+1})$$
$$q_{k+1} = q_k + \frac{1}{6}(v'_{k+1} + 2v''_{k+1} + 2v'''_{k+1} + v''''_{k+1})$$

- Explicit methods are often faster but less stable than implicit methods
- Stability and accuracy are different

The Laplace Operator

- All of our model equations used the Laplace operator
  - Laplace Equation $\Delta u = 0$
  - Heat Equation $\dot{u} = \Delta u$
  - Wave Equation $\ddot{u} = \Delta u$

- Unbelievably important object showing up everywhere across physics, geometry, signal processing, and more

- What does the Laplacian mean?
  - **Differential operator:** eats a function, spits out its 2nd derivative
  - What does that mean for a function: $u: \mathbb{R}^n \rightarrow \mathbb{R}$?
    - Divergence of gradient

$$\Delta u = \nabla \cdot \nabla u$$

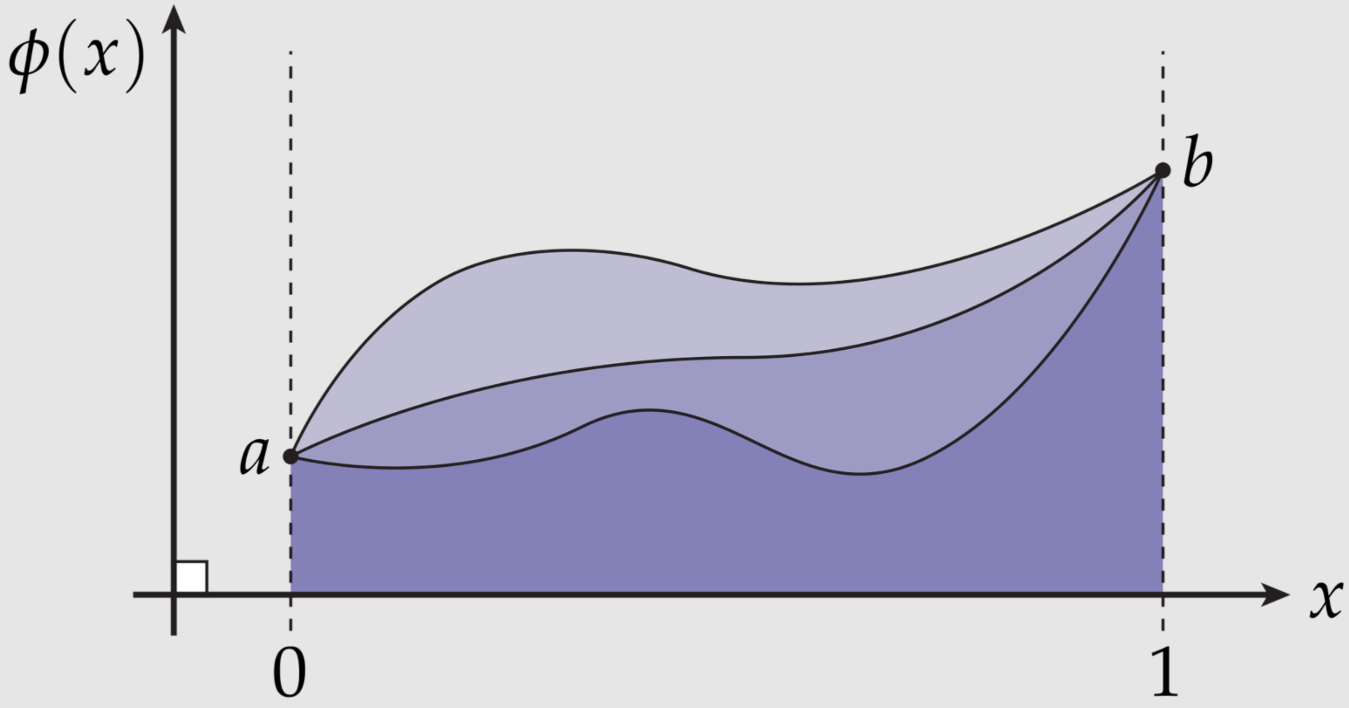
    - Sum of second derivatives

$$\Delta u = \frac{\partial u^2}{\partial x_1^2} + \cdots + \frac{\partial u^2}{\partial x_n^2}$$

    - Deviation from local average
    - …

Dirichlet Boundary Conditions

Dirichlet: boundary data always set to fixed values

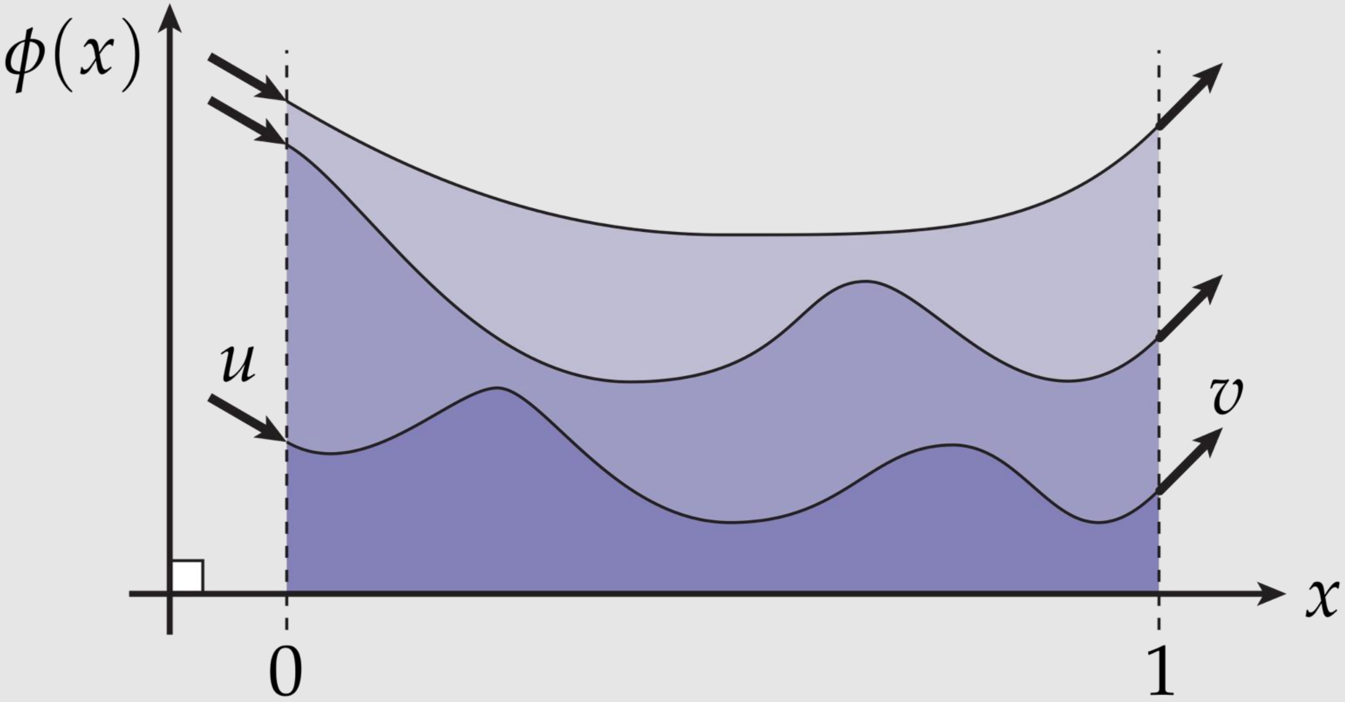


Example: $\phi(0) = a, \phi(1) = b$

Many possible functions interpolate values in between

Neumann Boundary Conditions

Neumann: specify derivatives across boundary



Example: $\phi'(0) = u \;\; , \phi'(1) = v$

Again, many possible functions

Discretizing The Laplacian

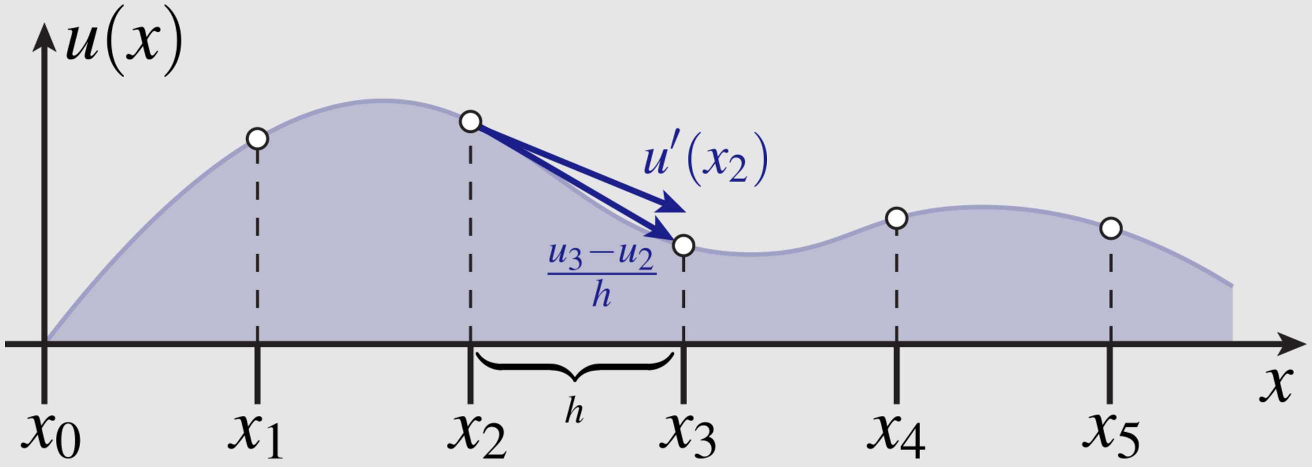
- Consider the Laplacian as a sum of second derivatives:

$$\Delta u = \frac{\partial u^2}{\partial x_1^2} + \cdots + \frac{\partial u^2}{\partial x_n^2}$$

- How do we compute this numerically?

- Consider a non-differentiable function with evaluated samples $x_0, x_1, \ldots$
  - The 1st-order derivative approximated is:

$$u'(x_i) \approx \frac{u_{i+1} - u_i}{h}$$



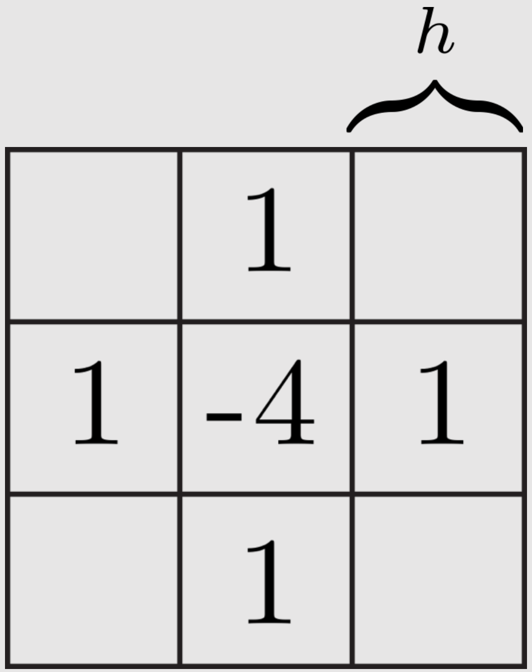
  - The 2nd-order derivative approximated is:

$$u''(x_i) \approx \frac{u'_i - u'_{i-1}}{h} \approx \frac{\left(\frac{u_{i+1} - u_i}{h}\right) - \left(\frac{u_i - u_{i-1}}{h}\right)}{h} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

- Known as the **finite difference** approach to PDEs

Discretizing The Laplacian

What if u is not a 1D function…

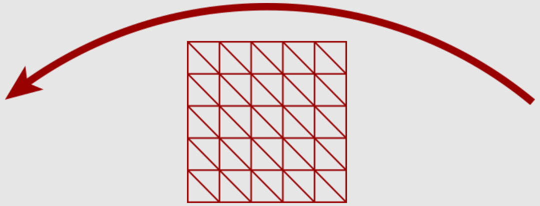


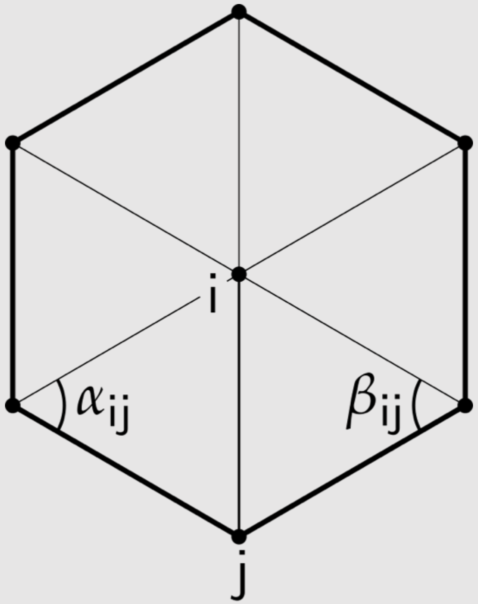
	1	
1	-4	1
	1	

$$\frac{4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}}{h^2}$$

[Grid]

If the mesh is a grid, equations become the same





$$\frac{1}{2}\sum_j (\cot \alpha_{ij} + \cot \beta_{ij})(u_j - u_i)$$

[Triangle Mesh]

Solving The Heat Equation

Heat equation tells us the Laplacian is equal to the first temporal derivative:

$$\dot{u} = \Delta u$$

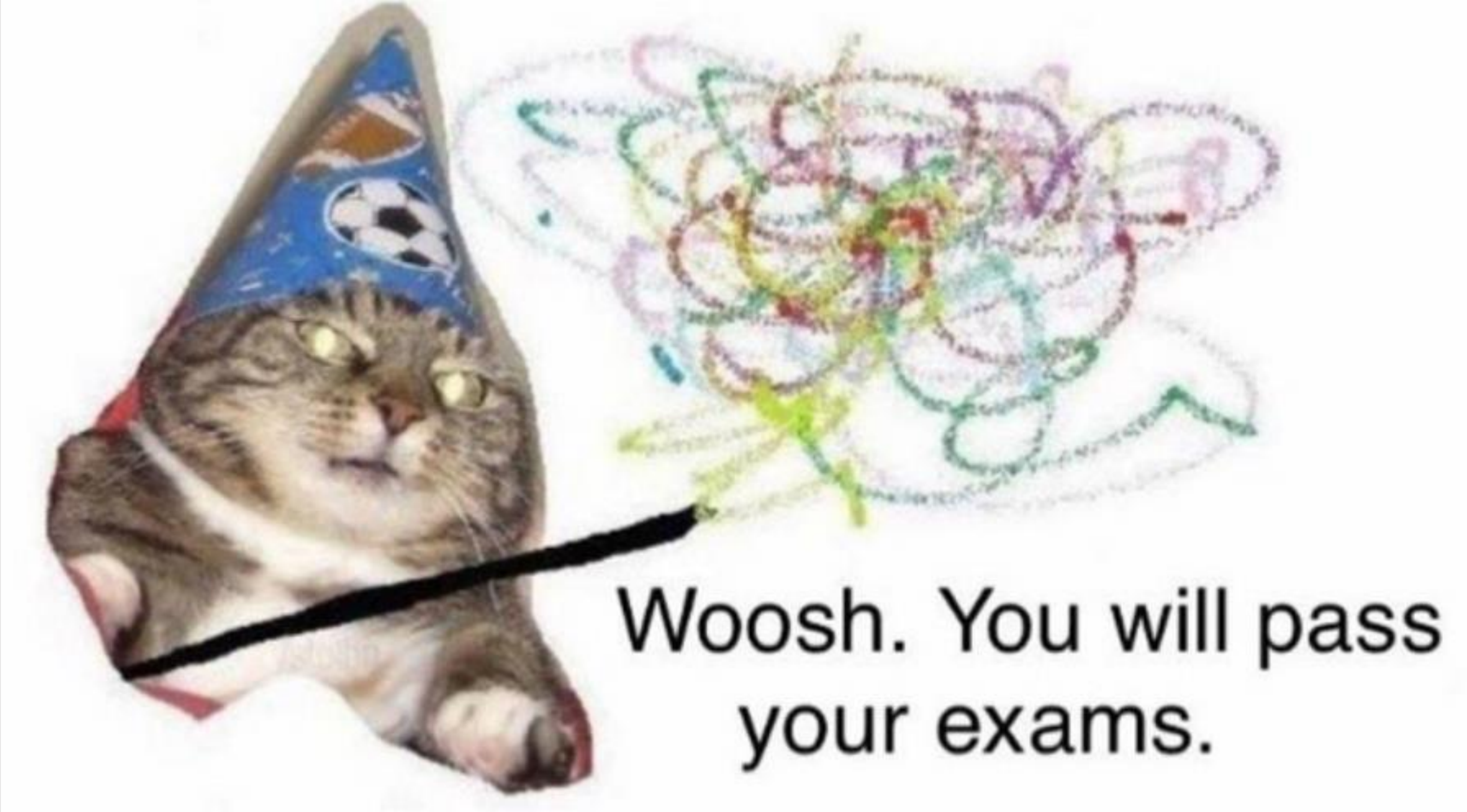
Compute the Laplacian approximately, e.g. using finite difference on a grid:

$$u_{i,j}^{k+1} = u^k + \frac{\tau}{h^2}(4u_{i,j}^k - u_{i+1,j}^k - u_{i-1,j}^k - u_{i,j+1}^k - u_{i,j-1}^k)$$

Propagate using the first temporal derivative Δu (Ex: forward Euler):

$$u^{k+1} = u^k + \tau \Delta u^k$$

Good Luck!



When the slides are over and you & your friends want to leave class but the professor keeps talking

How I sleep knowing I learned a lot from 15-362/662

Thank you for taking this course.