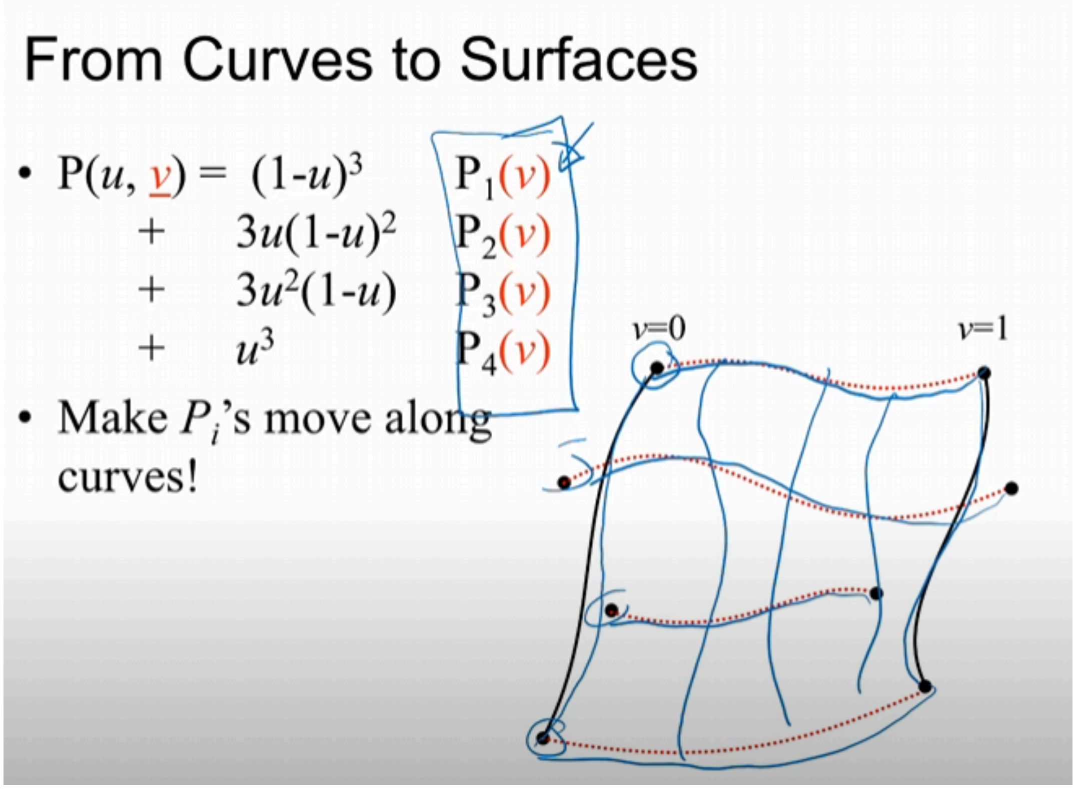
Introduction to Computer Graphics

- **L1: Introduction, Application**
  - Will Learn
    - Fundamentals of Computer Graphics Algorithms
    - Basics of real-time rendering: Basic OpenGL
    - C++
- **L2: Cubic Curves**
  - Hermite Basis
  - Cubic Blossom
  - Bernstein Polynomials
  - Cubic Control Polygon
  - Three Bases for Cubic Curves
    - Monomial basis
    - Hermite basis
    - bernstein basis
- **L3: Curves and Surfaces**
  - Curves
    - Order of Continuity
      - C0 = continuous
      - G1 = geometric continuity
        - tangents align at the seam
      - C1 = paraMetric continuity
        - same velocity at the seam
      - C2 = curvature continuity
        - tangens and their derivatives are the same
    - Cubic B-Splines
      - Automatically C2
    - Converting Between Bezier & BSpline
  - Surfaces
    - Trangle Meshes
      - Simple, rendered directly
      - not smooth, need many trangles to smooth
    - Tensor Product Splines

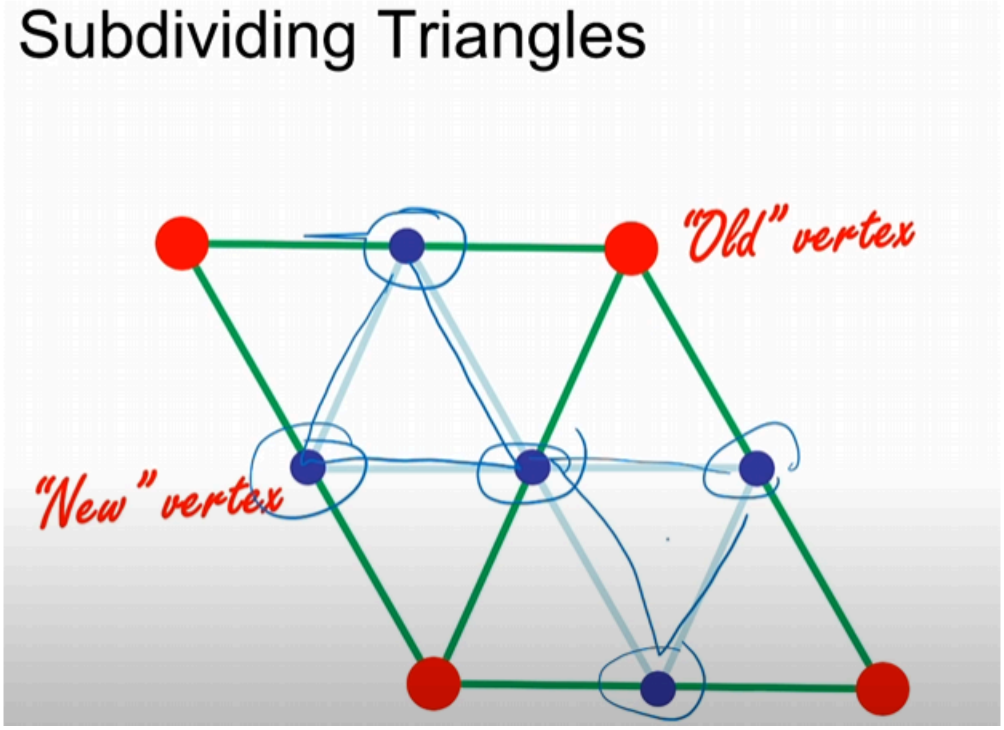
    - From Curves to Surfaces



From Curves to Surfaces

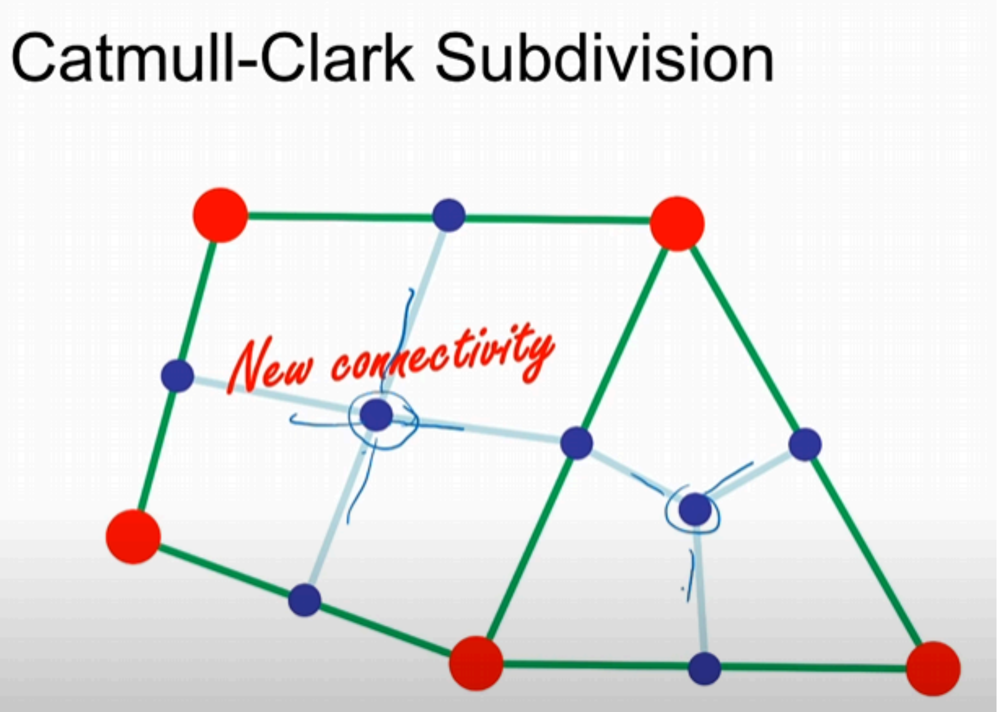
- $$\begin{aligned} P(u, v) = {} & (1-u)^3 && P_1(v) \\ + {} & 3u(1-u)^2 && P_2(v) \\ + {} & 3u^2(1-u) && P_3(v) \\ + {} & u^3 && P_4(v) \end{aligned}$$
- Make P_i's move along curves!

- Subdivision Surfaces
    - Corner Cutting
        - Subdividing Triangles



Subdividing Triangles

- Catmull-Clark Cubdivision

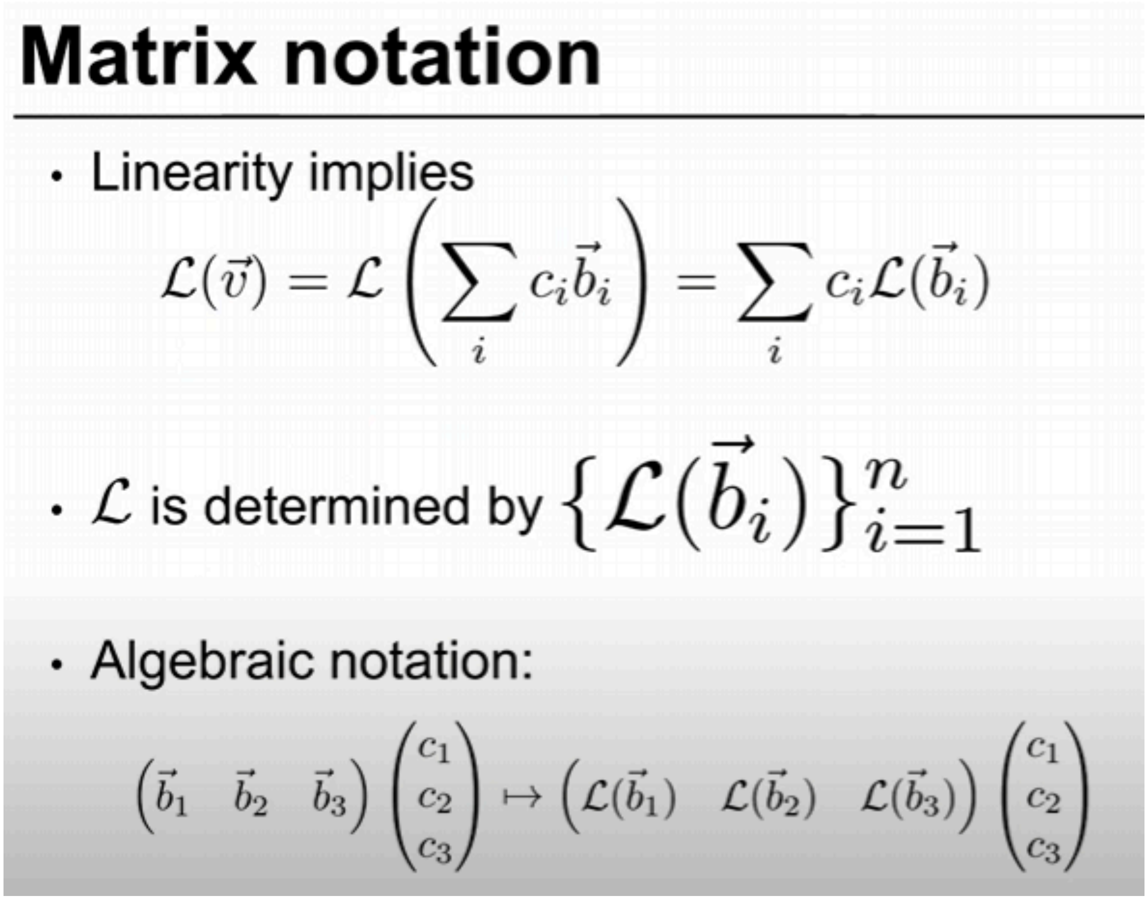


Catmull-Clark Subdivision

- Advantages
  - Arbitrary topology
  - Smooth at boundaries
  - Level of detail, scalable
  - Simple representation
  - Numerical stability, well-behaved meshes
  - Code simplicity
- Disadvantage
  - Procedural definition
  - Not parametric
  - Tricky at special vertices

- Implicit Surfaces
  - Surface defined implicitly by a function
    - f(x, y, z) = 0 (on surface)
    - f(x, y, z) < 0 (inside surface)
    - f(x, y, z) > 0 (outside surface)
  - Pros:
    - Efficient check whether point is inside
    - Efficient Bollean operations
    - Can handle weird topology for animation
    - Easy to do sketchy modeling
  - Cons:

      - Hard to generate points on the surface
    - Procedural
- **L4: Coordinates and transformations**
  - Linear algebra notation
    - Matrix notation



Matrix notation

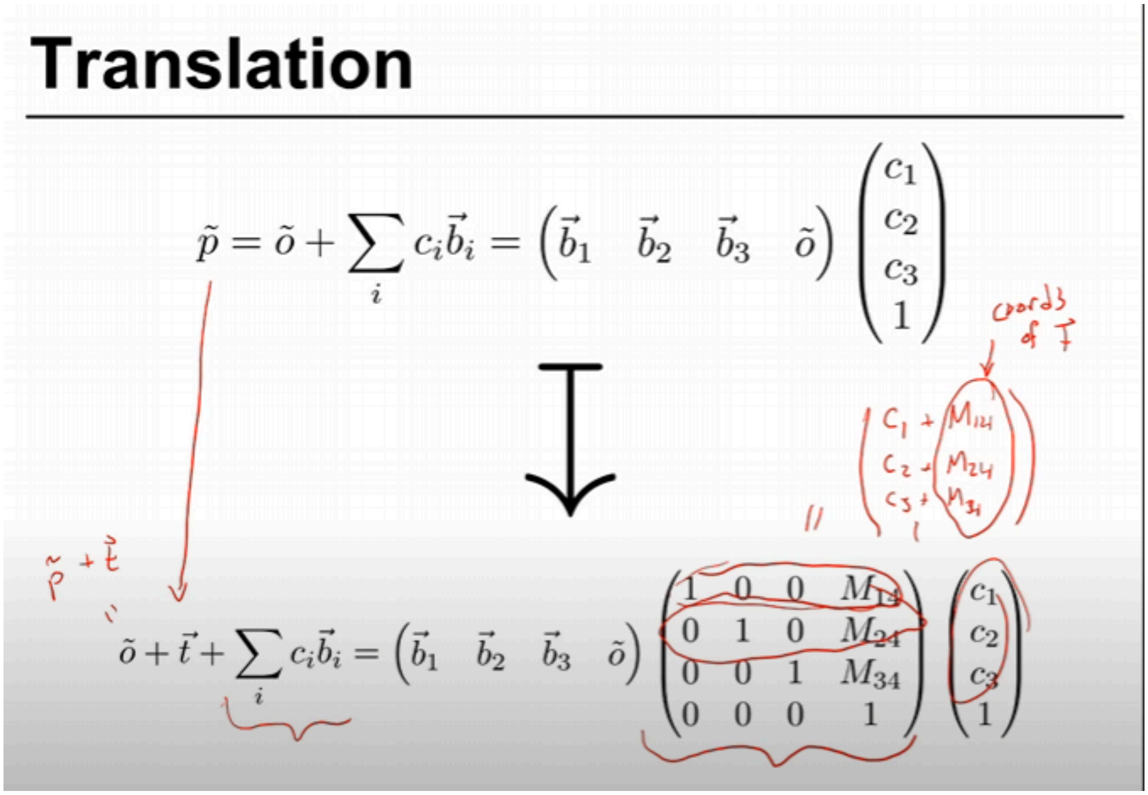
- Linearity implies

$$\mathcal{L}(\vec{v}) = \mathcal{L}\left(\sum_i c_i \vec{b}_i\right) = \sum_i c_i \mathcal{L}(\vec{b}_i)$$

- $\mathcal{L}$ is determined by $\left\{\mathcal{L}(\vec{b}_i)\right\}_{i=1}^{n}$
- Algebraic notation:

$$\begin{pmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{L}(\vec{b}_1) & \mathcal{L}(\vec{b}_2) & \mathcal{L}(\vec{b}_3) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

- Translation



Translation

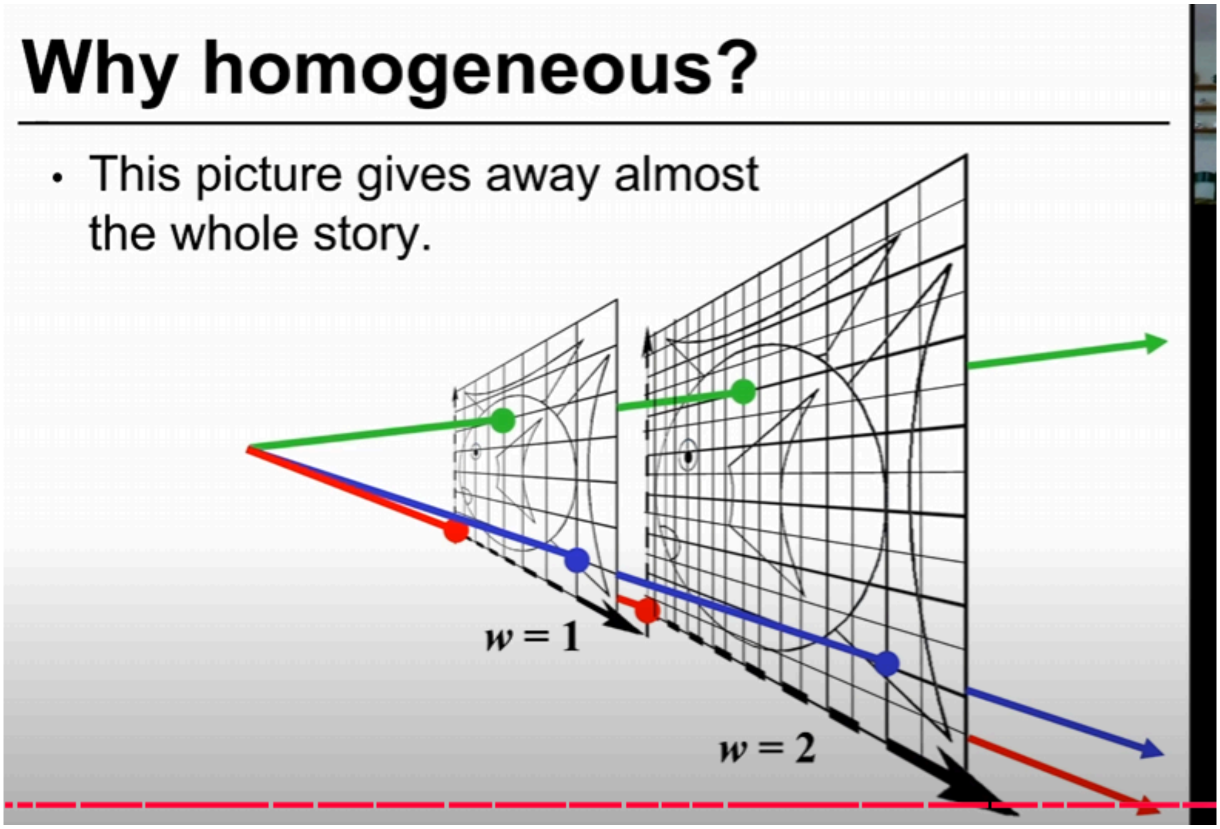
$$\tilde{p} = \tilde{o} + \sum_i c_i \vec{b}_i = \begin{pmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{pmatrix}$$

$$\downarrow$$

$$\underbrace{\tilde{o} + \vec{t} + \sum_i c_i \vec{b}_i}_{\tilde{p} + \vec{t}} = \begin{pmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{pmatrix} \underbrace{\begin{pmatrix} 1 & 0 & 0 & M_{14} \\ 0 & 1 & 0 & M_{24} \\ 0 & 0 & 1 & M_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{pmatrix}}_{= \begin{pmatrix} c_1 + M_{14} \\ c_2 + M_{24} \\ c_3 + M_{34} \\ 1 \end{pmatrix}}$$

coords of $\vec{t}$

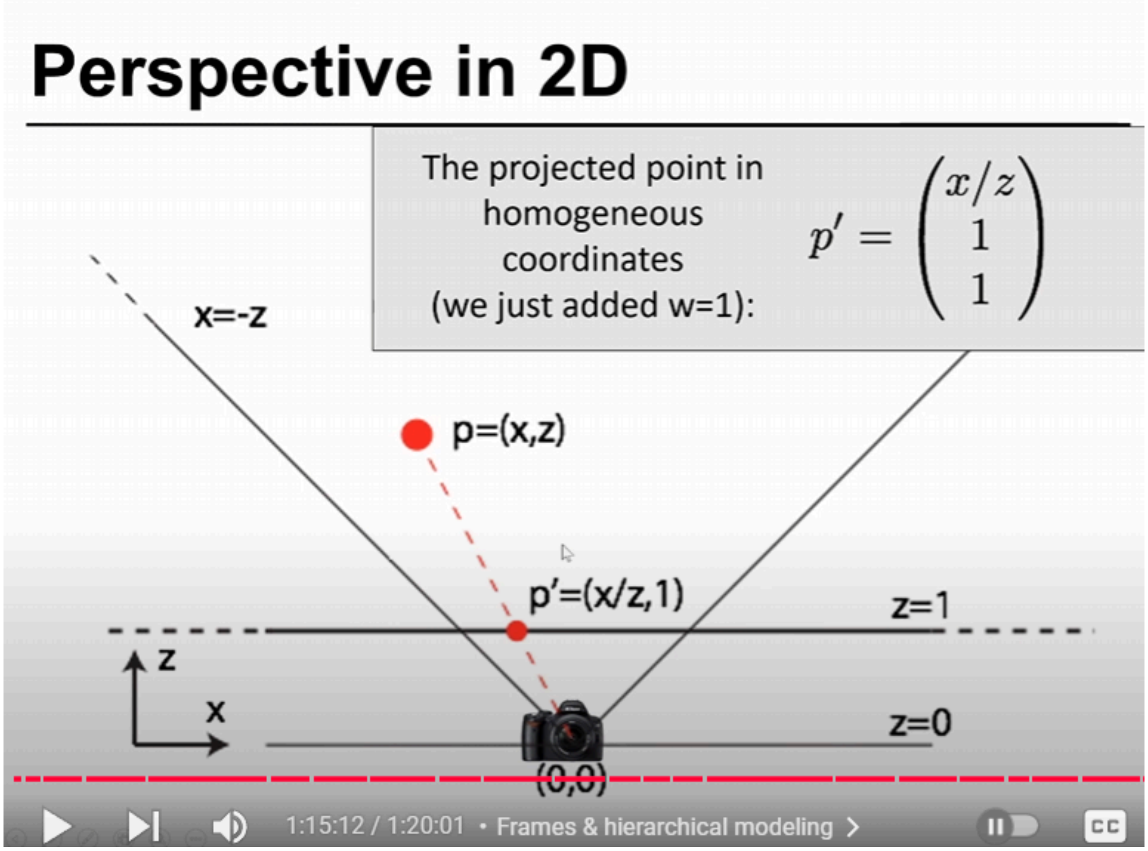
- Homogeneous Coordination



Why homogeneous?

- This picture gives away almost the whole story.

- For perspective projection



Perspective in 2D

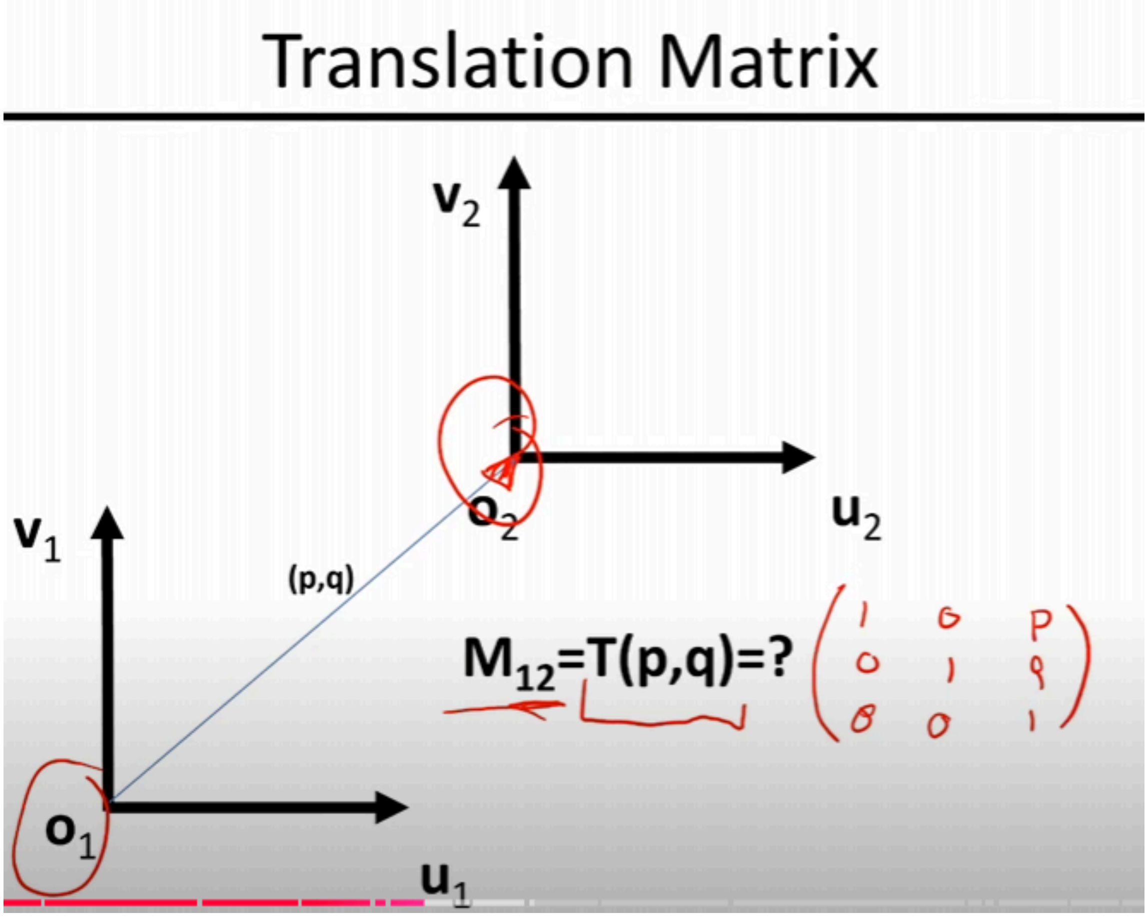
The projected point in homogeneous coordinates (we just added w=1):

$$p' = \begin{pmatrix} x/z \\ 1 \\ 1 \end{pmatrix}$$

- For ray tracing algorithm

- **L5: Hierachical modeling and scene graphs**
  - Coordinate System transformation

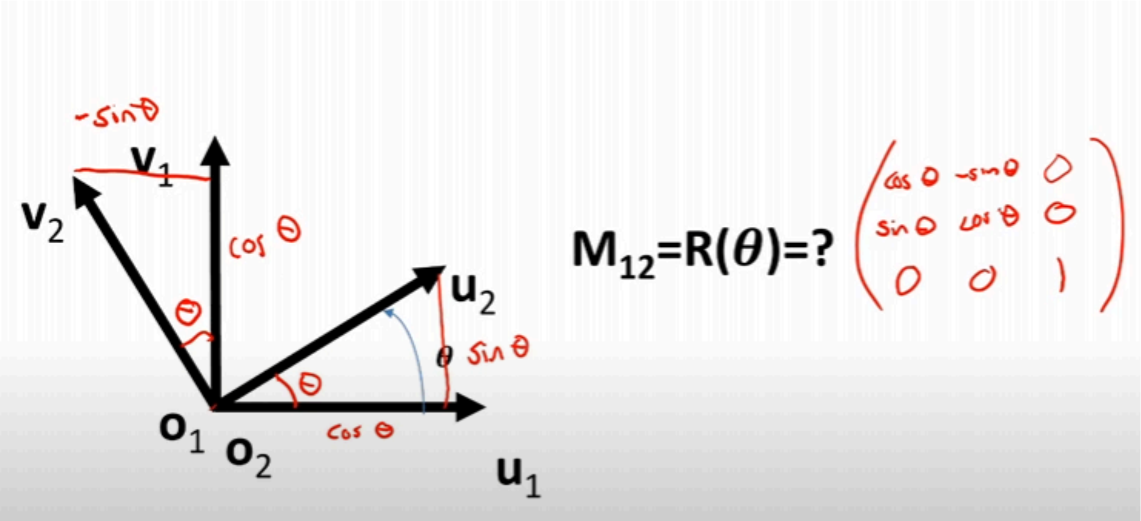
- Translation Matrix



Translation Matrix

$$M_{12}=\underbrace{T(p,q)}=? \begin{pmatrix}1 & 0 & p\\ 0 & 1 & q\\ 0 & 0 & 1\end{pmatrix}$$

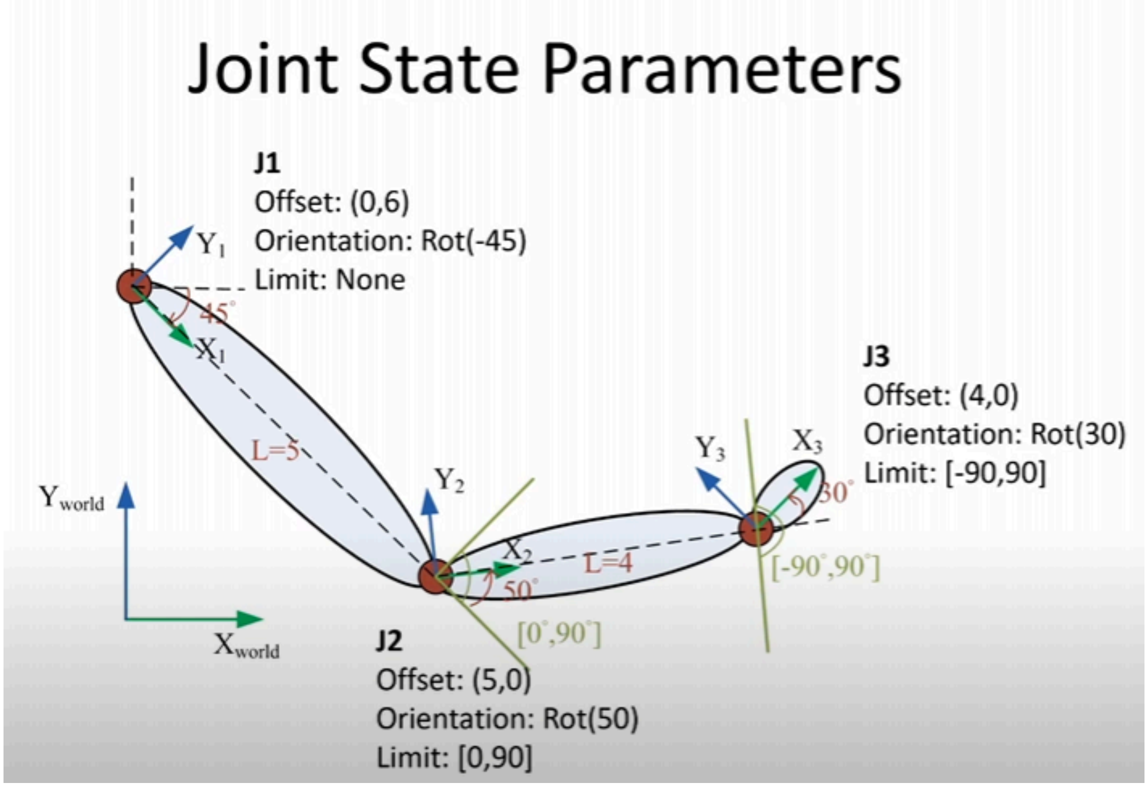
- Rotation Matrix



$$M_{12}=R(\theta)=? \begin{pmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{pmatrix}$$

- Forward and inverse kinematics
  - Joints

- Joint State Parameters



Joint State Parameters

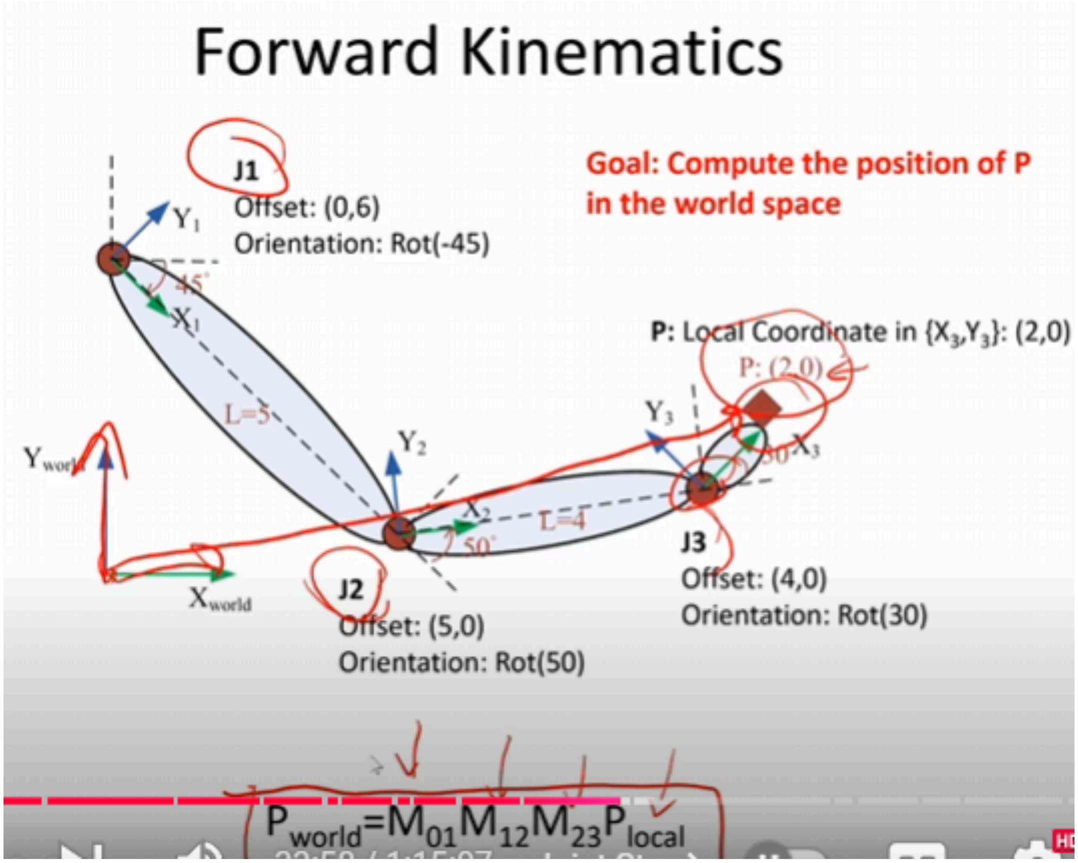
J1
Offset: (0,6)
Orientation: Rot(-45)
Limit: None

J3
Offset: (4,0)
Orientation: Rot(30)
Limit: [-90,90]

J2
Offset: (5,0)
Orientation: Rot(50)
Limit: [0,90]

  - Offset
  - Orientation
  - Limit

- Forward Kinematics



Forward Kinematics

J1
Offset: (0,6)
Orientation: Rot(-45)

Goal: Compute the position of P in the world space

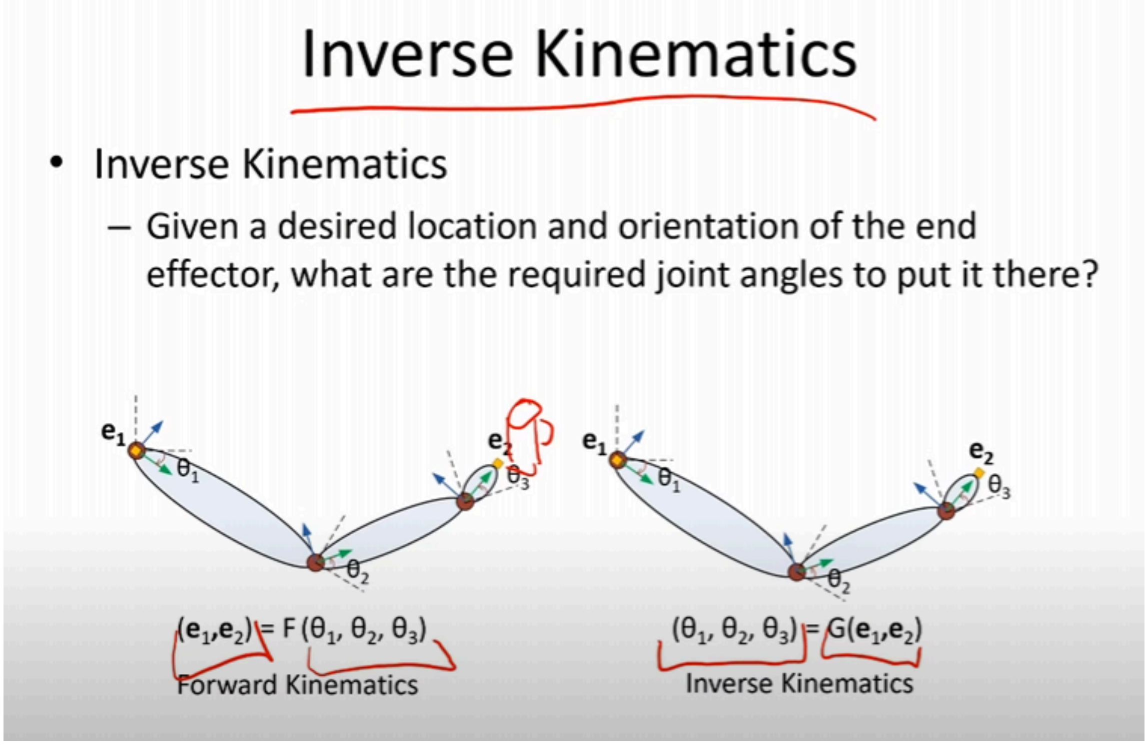
P: Local Coordinate in $\{X_3,Y_3\}$: (2,0)

J2
Offset: (5,0)
Orientation: Rot(50)

J3
Offset: (4,0)
Orientation: Rot(30)

$$P_{world}=M_{01}M_{12}M_{23}P_{local}$$

        - Inverse Kinematics



Inverse Kinematics

- Inverse Kinematics
  - Given a desired location and orientation of the end effector, what are the required joint angles to put it there?

$(e_1,e_2) = F(\theta_1, \theta_2, \theta_3)$

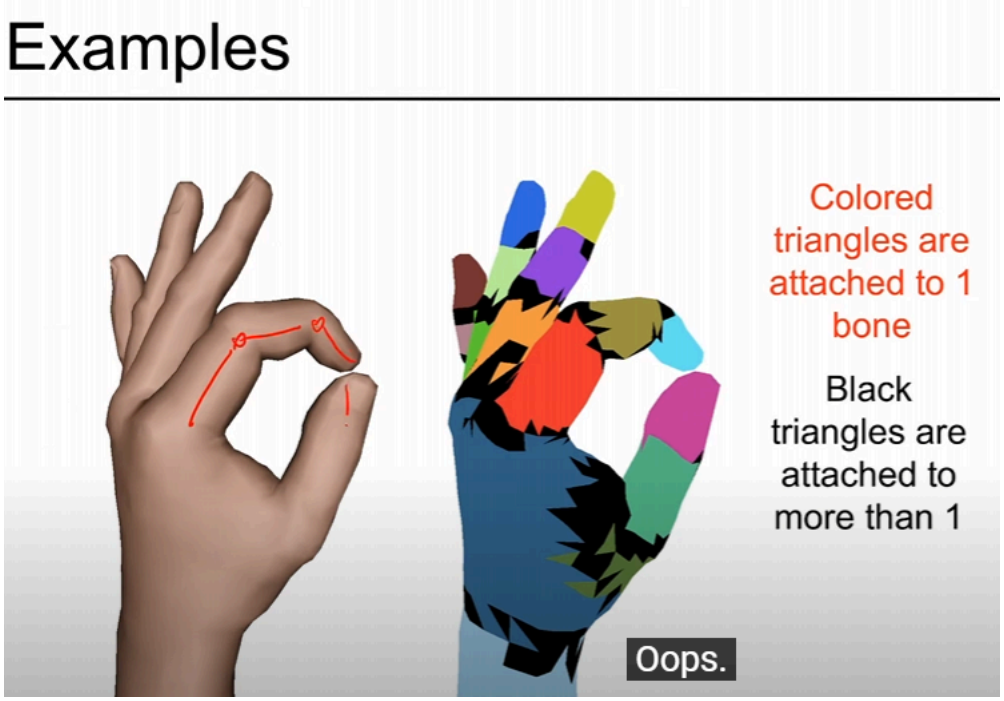
Forward Kinematics

$(\theta_1, \theta_2, \theta_3) = G(e_1,e_2)$

Inverse Kinematics

    - Hierarchical tree and scene graph
- **L6: Introduction to Animation and Skinning/Enveloping**
    - Types of Animation:
        - Keyframing
        - Procedural
            - Express animation as as funciton
        - Physicial Based
    - Animation Controls
        - Forward Kinematic
        - Inverse Kinematic
        - Skinning Characters
            - Bind Skin vertices to bone
                - Motion Capture
                    - Retargeting
    - Character Animation
        - Skinning/Enveloping
            - Skeletal subspace deformation (SSD)

- Bind vertice to 1 bone or multiple bone



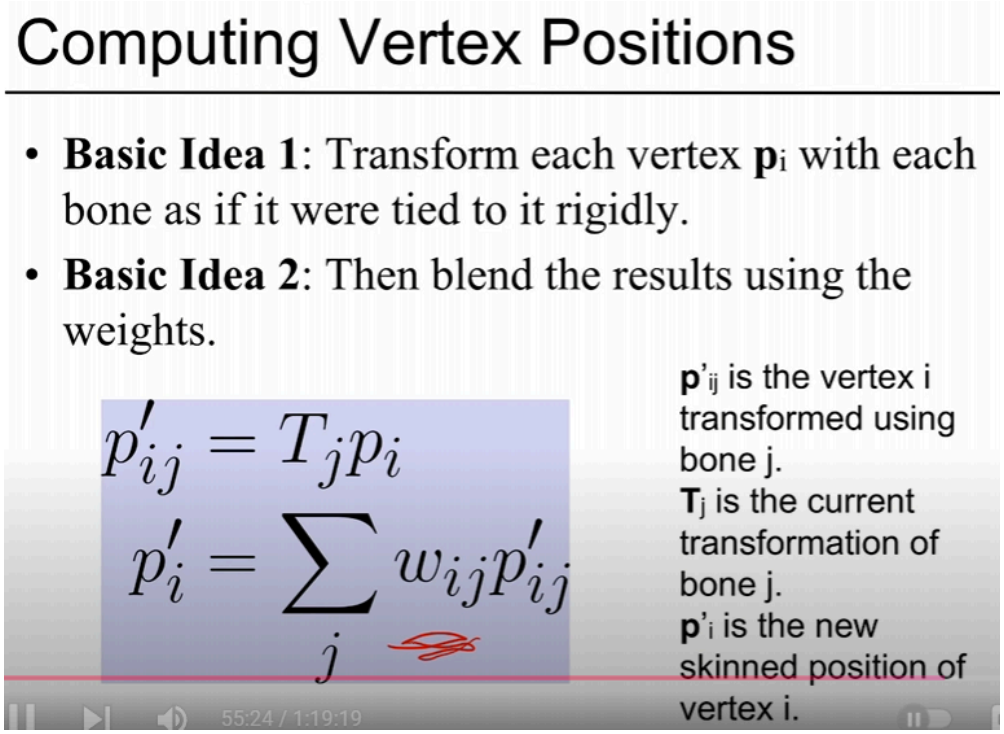
Examples

Colored triangles are attached to 1 bone

Black triangles are attached to more than 1

Oops.

- Vertex Weights
- Linear Blend Skinning



Computing Vertex Positions

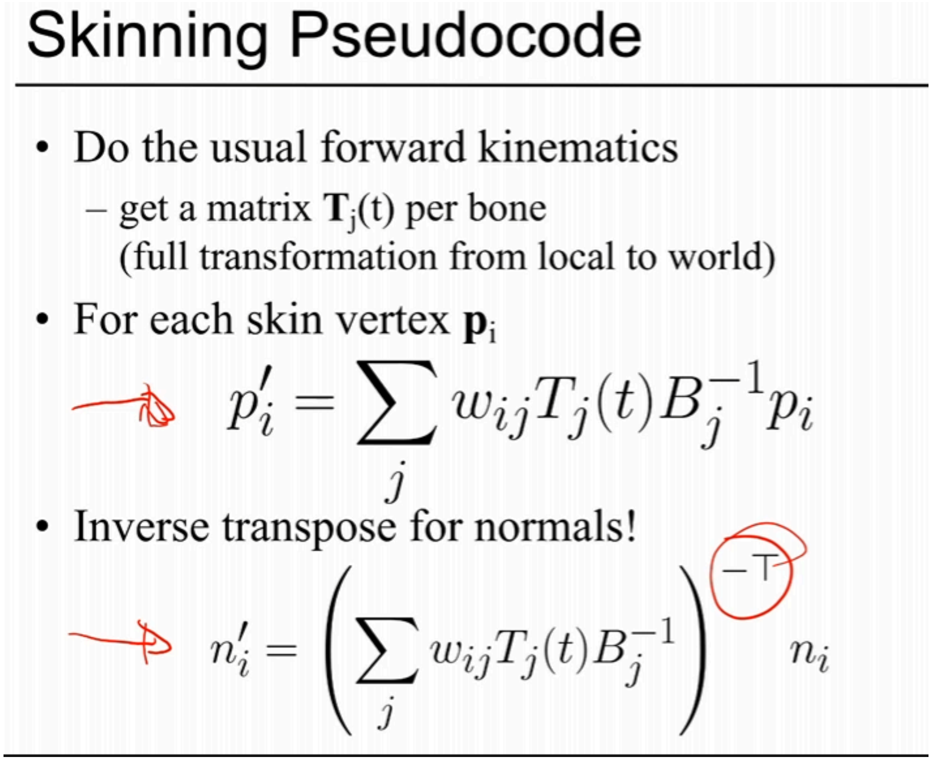
- **Basic Idea 1**: Transform each vertex $\mathbf{p}_i$ with each bone as if it were tied to it rigidly.
- **Basic Idea 2**: Then blend the results using the weights.

$$p'_{ij} = T_j p_i$$

$$p'_i = \sum_j w_{ij} p'_{ij}$$

$\mathbf{p}'_{ij}$ is the vertex i transformed using bone j.
$\mathbf{T}_j$ is the current transformation of bone j.
$\mathbf{p}'_i$ is the new skinned position of vertex i.

- Bind Pose and weight



Skinning Pseudocode

- Do the usual forward kinematics
  - get a matrix $\mathbf{T}_j(t)$ per bone (full transformation from local to world)
- For each skin vertex $\mathbf{p}_i$

$$p_i' = \sum_j w_{ij} T_j(t) B_j^{-1} p_i$$

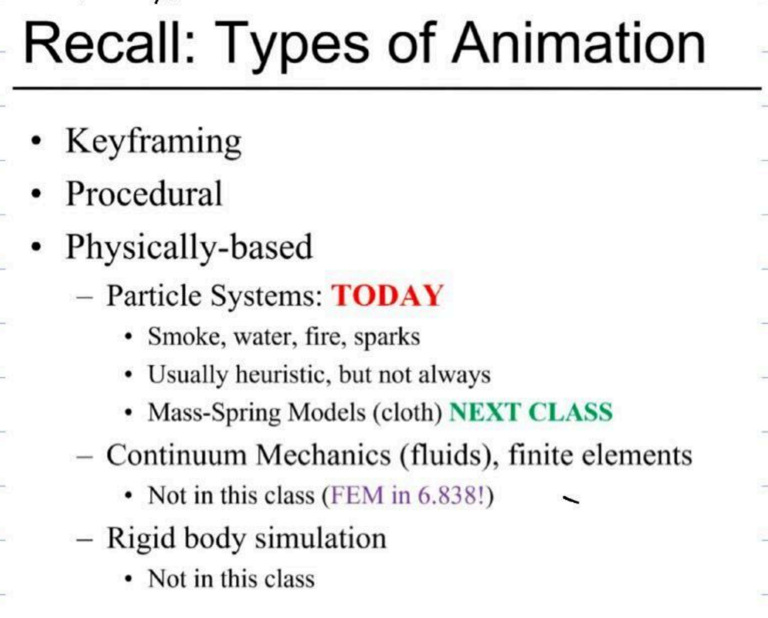
- Inverse transpose for normals!

$$n_i' = \left(\sum_j w_{ij} T_j(t) B_j^{-1}\right)^{-T} n_i$$

- L7 Particle System and ODE

L7: Particle System and ODEs

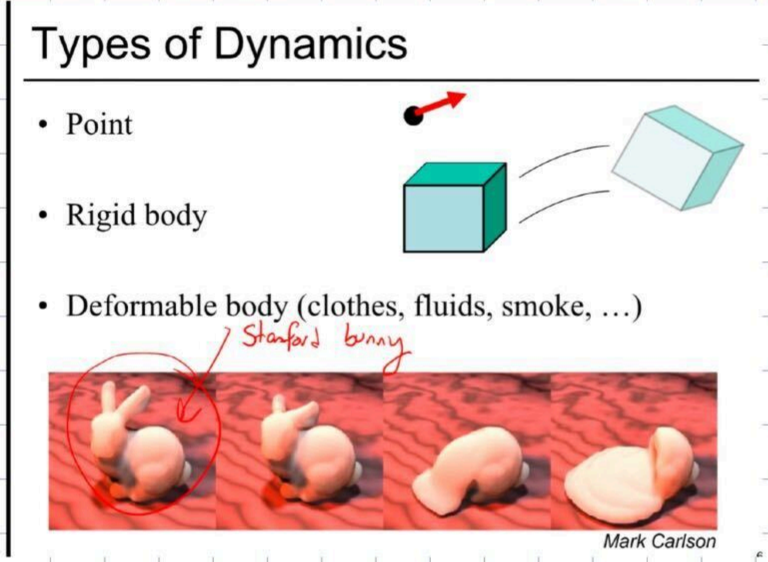
- Type of Animation : Physical Based
  - Particle System



Recall: Types of Animation

- Keyframing
- Procedural
- Physically-based
  - Particle Systems: **TODAY**
    - Smoke, water, fire, sparks
    - Usually heuristic, but not always
    - Mass-Spring Models (cloth) **NEXT CLASS**
  - Continuum Mechanics (fluids), finite elements
    - Not in this class (FEM in 6.838!)
  - Rigid body simulation
    - Not in this class

- Types of Dynamics



Types of Dynamics

- Point
- Rigid body
- Deformable body (clothes, fluids, smoke, …)

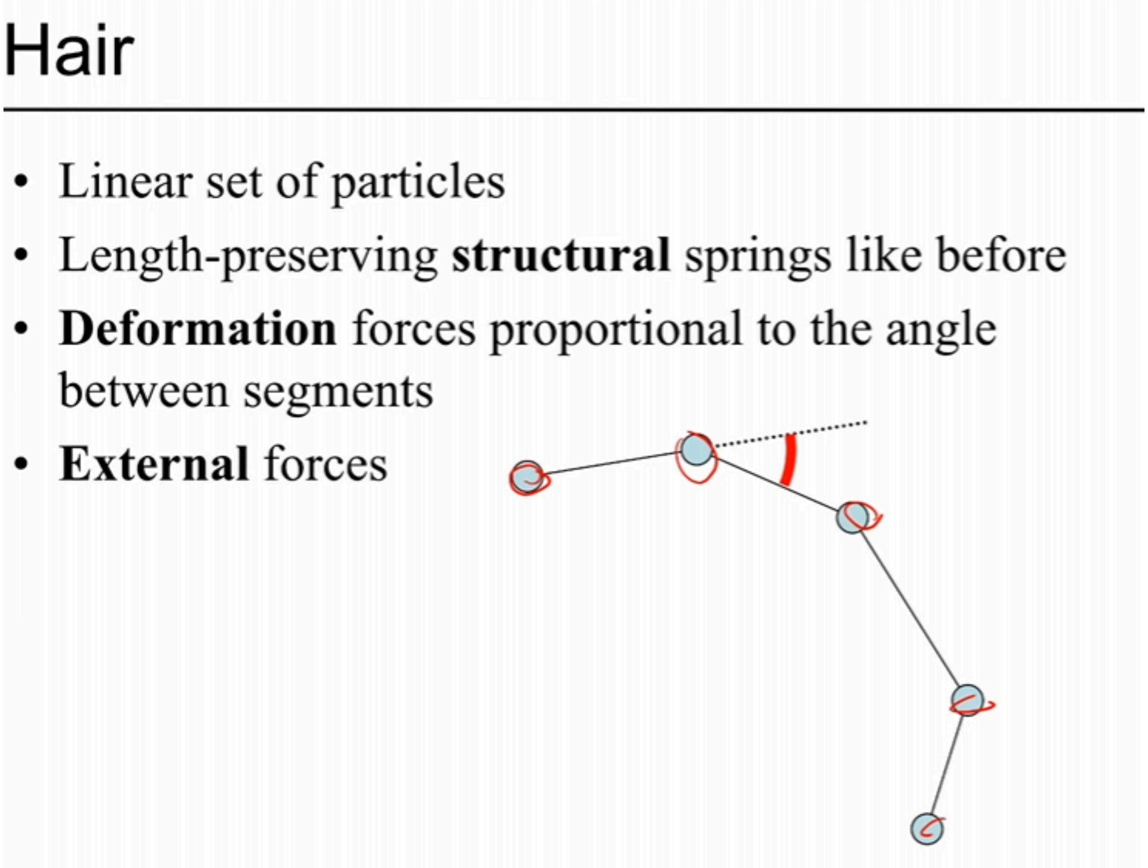
Stanford bunny

Mark Carlson

- Particle System
  - Emitter
  - Force
  - ODEs
  - Randomness

e.g. Flocking, Smoke, Fire, Water, Spark

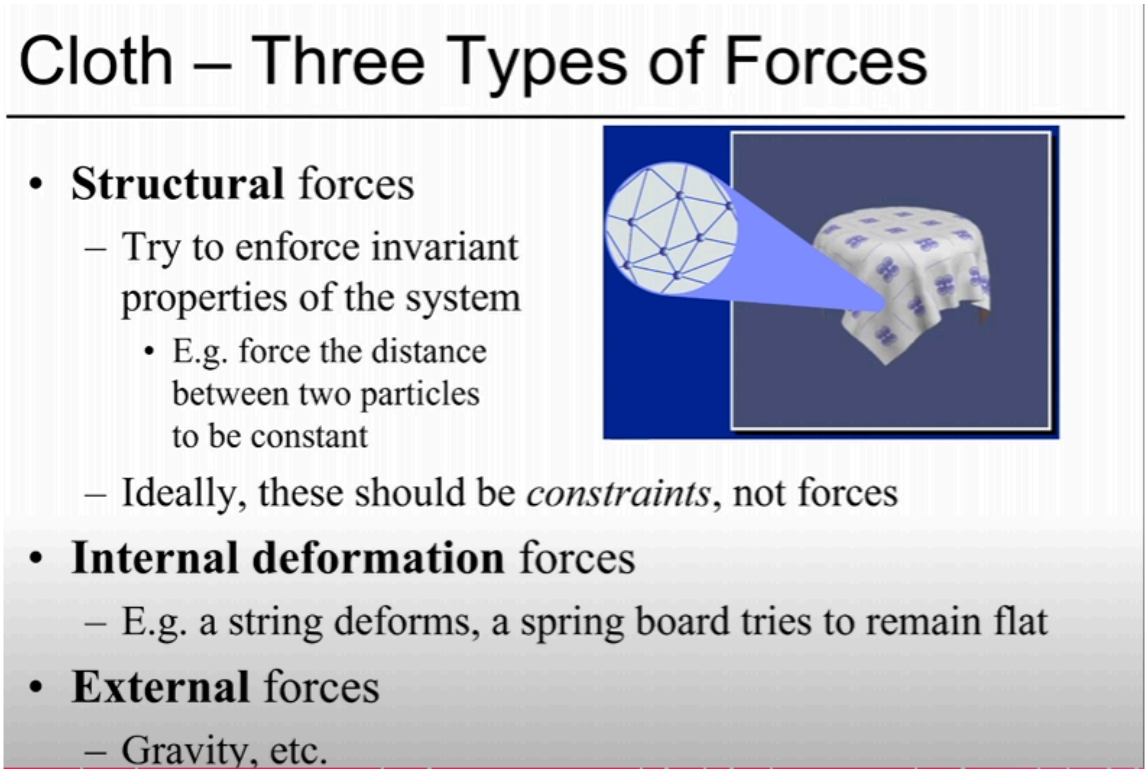
- **L8: More ODEs, mass-spring modeling, cloth simulation**
    - Euler's Method: Not Always Stable
        - Midpoint
        - Trapezoid
        - Runge-Kutta (RK4) Integrator
    - Mass-Spring Modeling
        - Hair



Hair

- Linear set of particles
- Length-preserving **structural** springs like before
- **Deformation** forces proportional to the angle between segments
- **External** forces

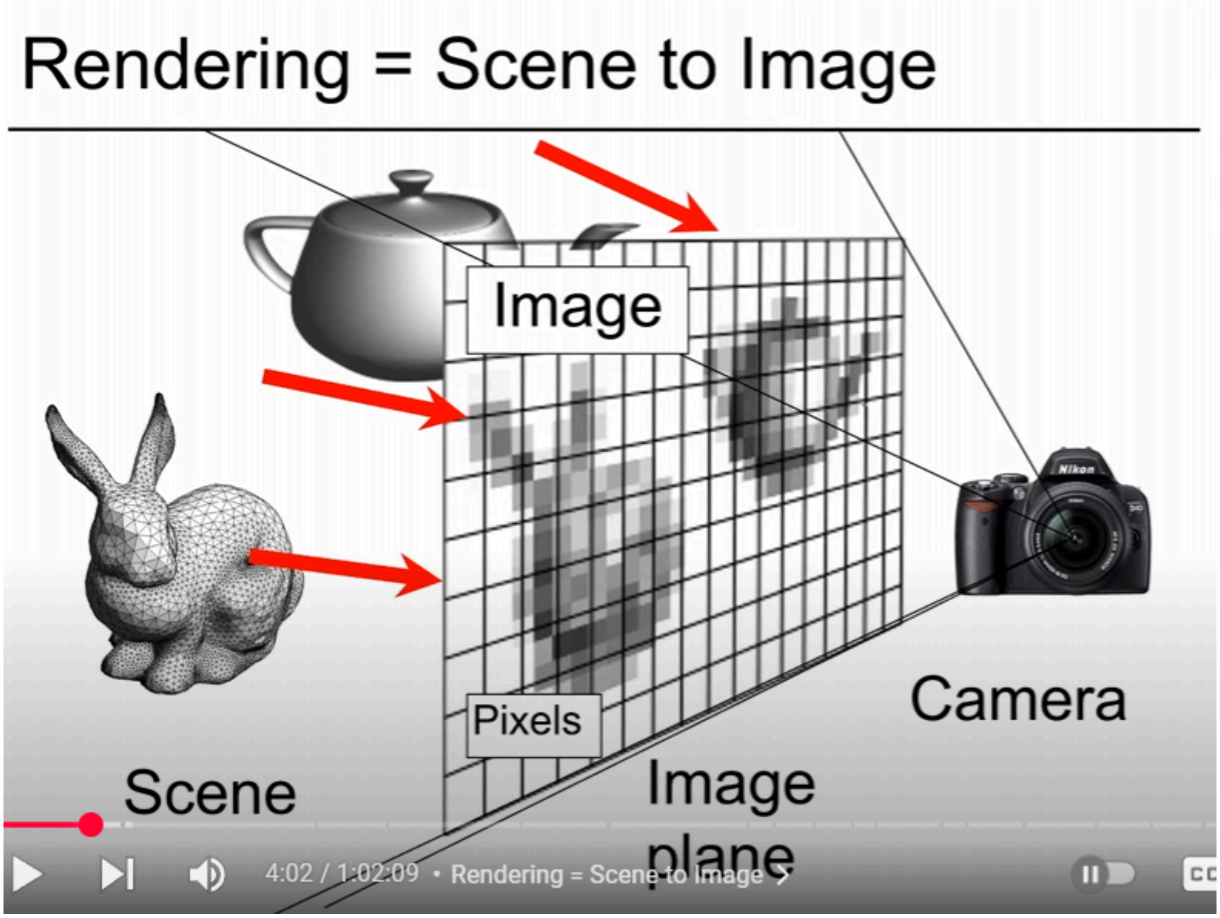
- Mass-Spring Cloth



Cloth – Three Types of Forces

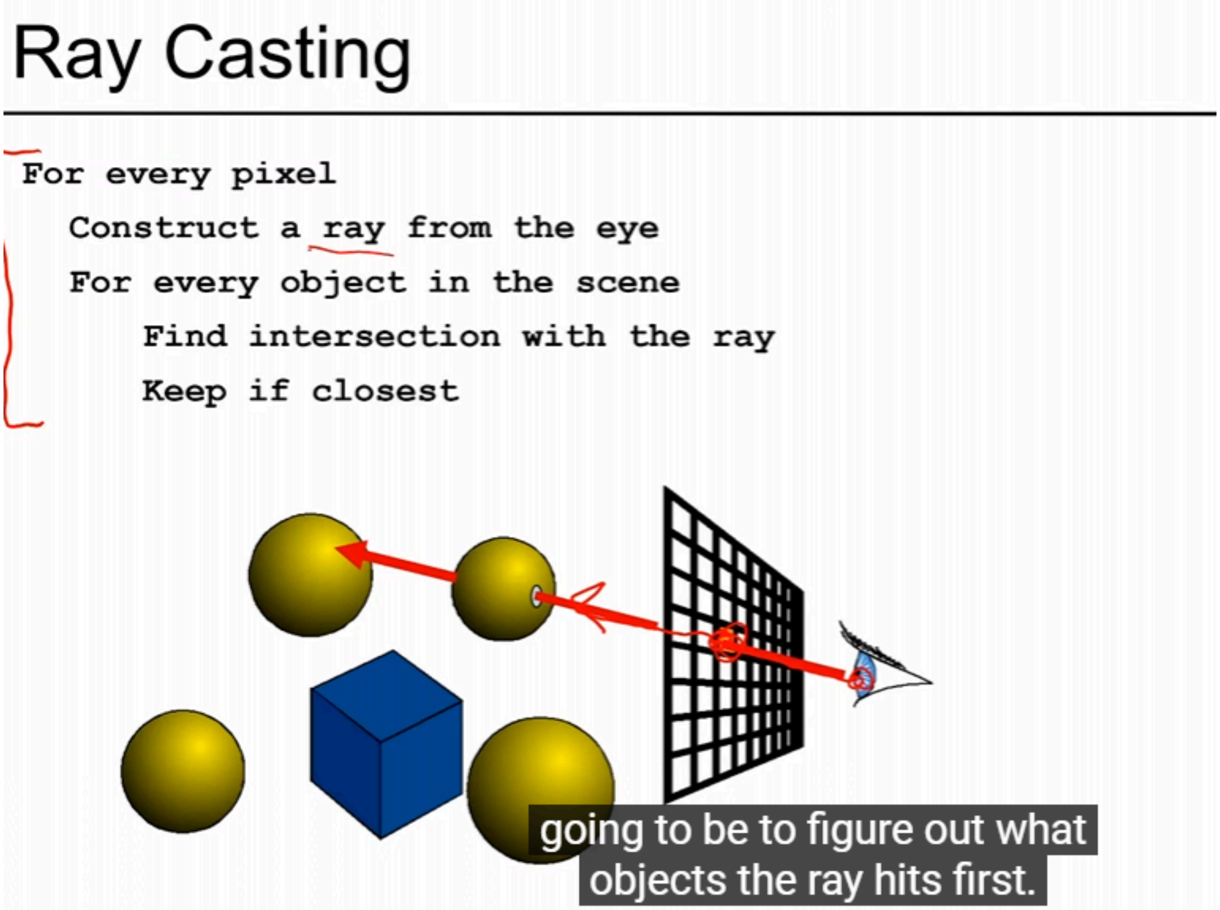
- **Structural** forces
  - Try to enforce invariant properties of the system
    - E.g. force the distance between two particles to be constant
  - Ideally, these should be *constraints*, not forces
- **Internal deformation** forces
  - E.g. a string deforms, a spring board tries to remain flat
- **External** forces
  - Gravity, etc.

- **L9: Introduction to Rendering, Ray Casting**
  - Rendering



Rendering = Scene to Image

- Ray Casting

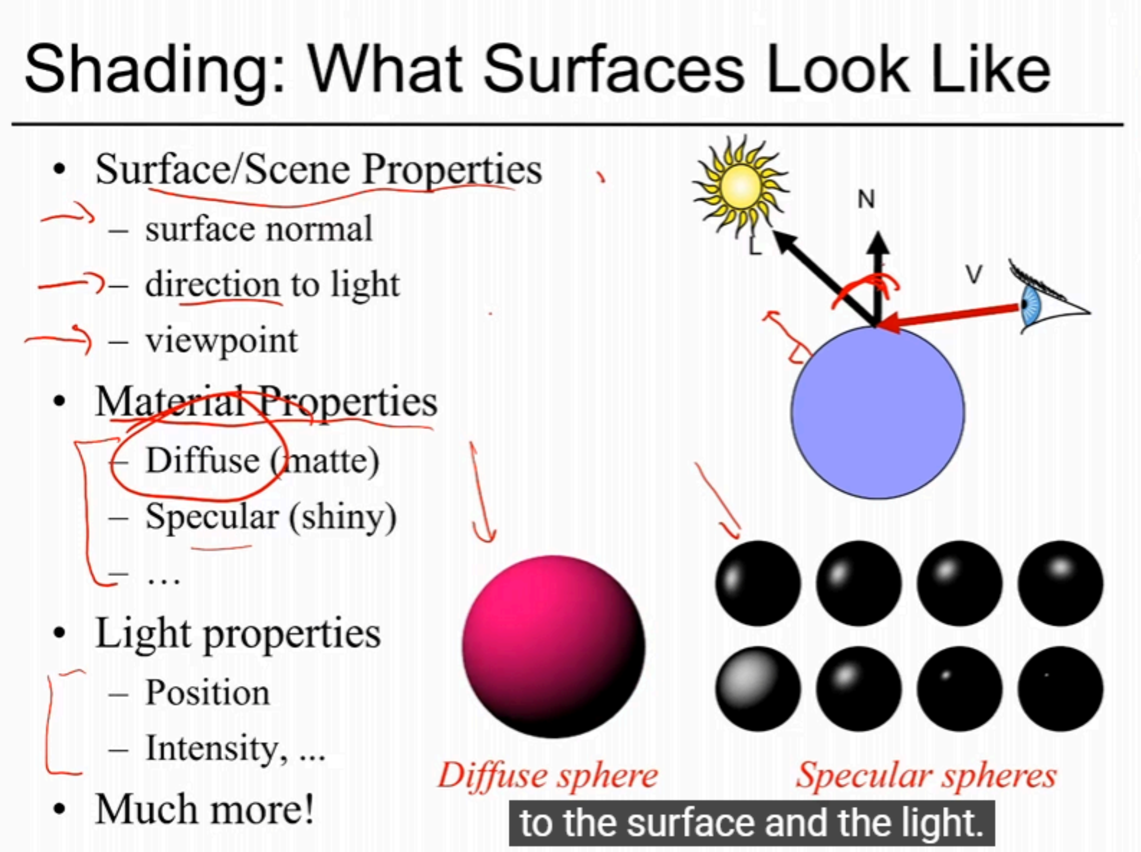


Ray Casting

```
For every pixel
    Construct a ray from the eye
    For every object in the scene
        Find intersection with the ray
        Keep if closest
```

going to be to figure out what objects the ray hits first.

  - Shading



Shading: What Surfaces Look Like

- Surface/Scene Properties
  - surface normal
  - direction to light
  - viewpoint
- Material Properties
  - Diffuse (matte)
  - Specular (shiny)
  - …
- Light properties
  - Position
  - Intensity, ...
- Much more!

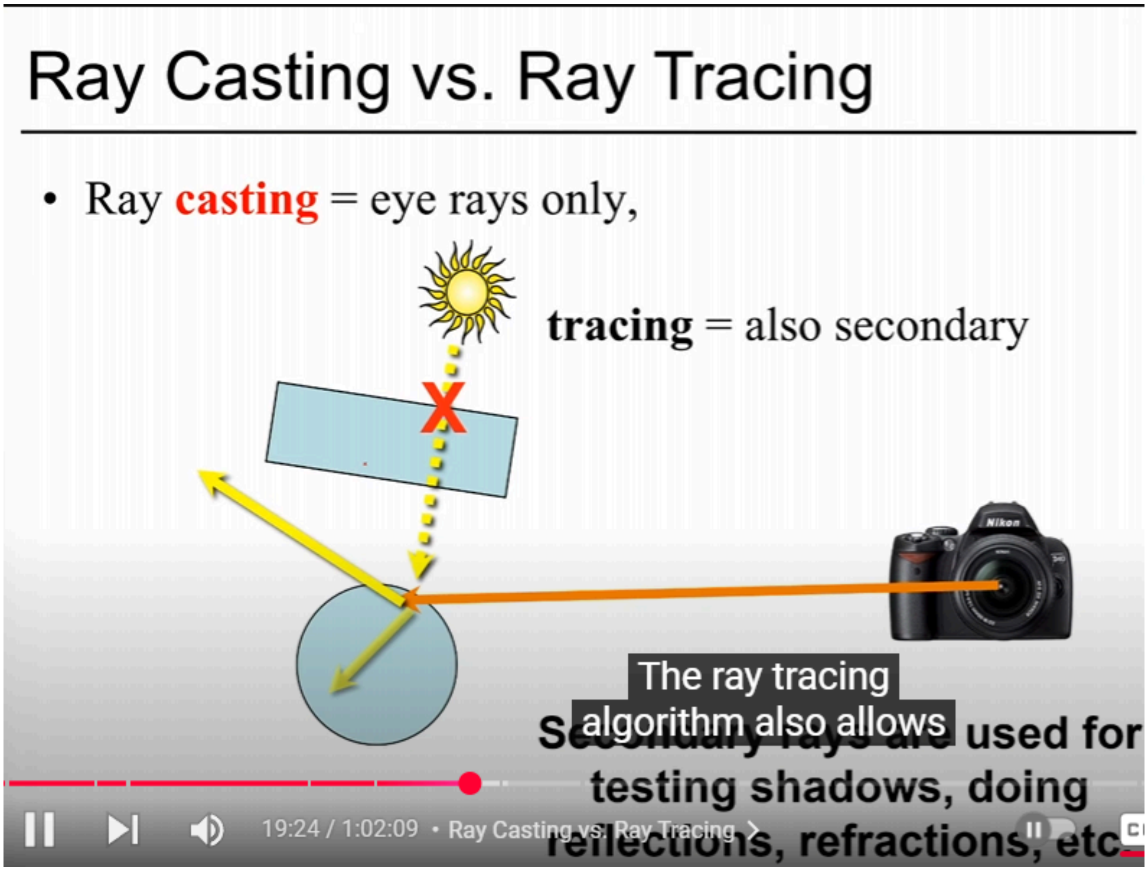
Diffuse sphere

Specular spheres

to the surface and the light.

    - Surface/Scene Properties

  - Material Properties
  - Light Properties
- Ray Casting vs. Ray Tracing



Ray Casting vs. Ray Tracing

- Ray **casting** = eye rays only,

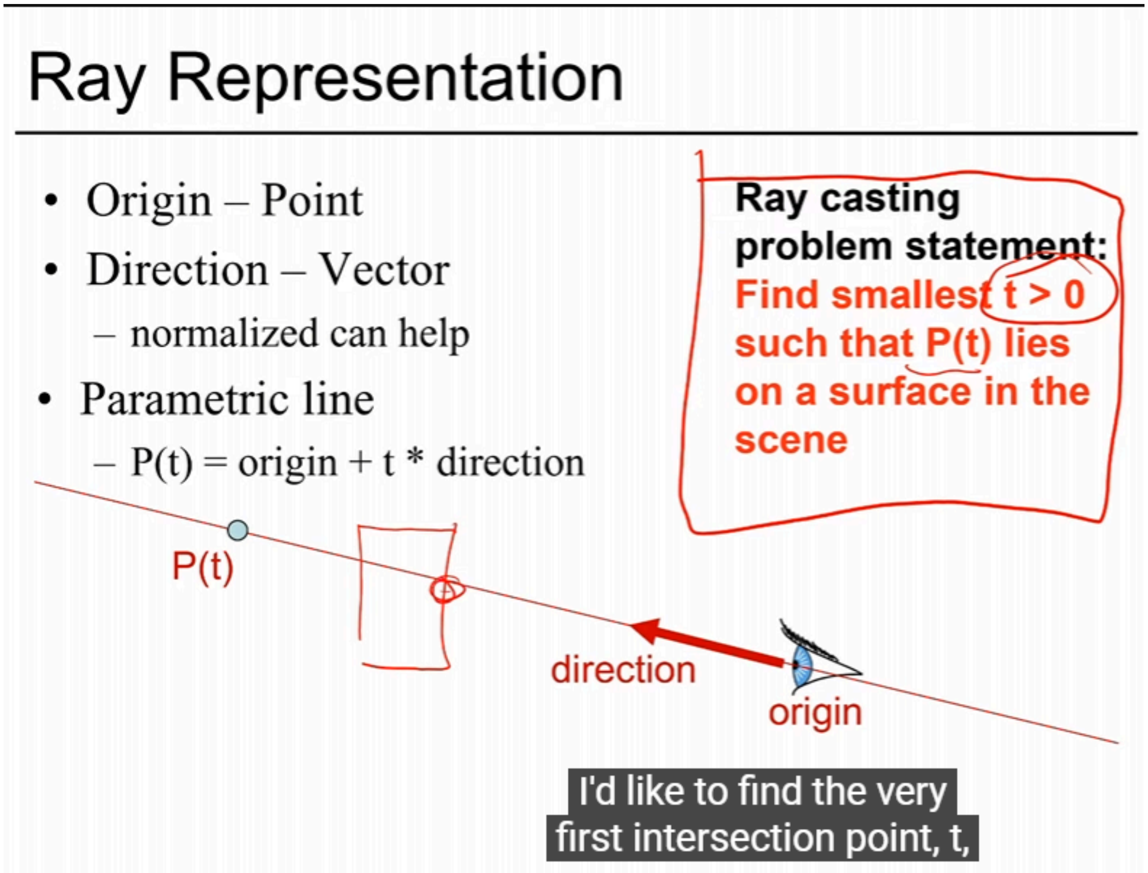
tracing = also secondary

Secondary rays are used for testing shadows, doing reflections, refractions, etc.

The ray tracing algorithm also allows

19:24 / 1:02:09 • Ray Casting vs. Ray Tracing >

- Ray Representation



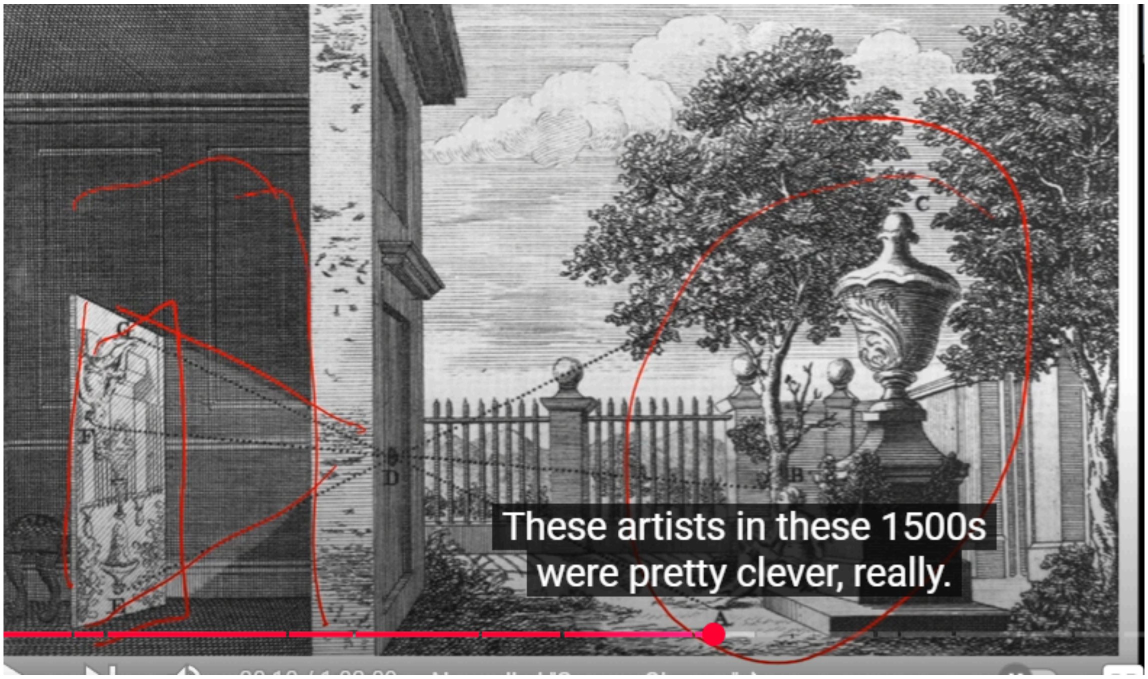
Ray Representation

- Origin – Point
- Direction – Vector
  - normalized can help
- Parametric line
  - P(t) = origin + t * direction

Ray casting problem statement:
Find smallest t > 0 such that P(t) lies on a surface in the scene

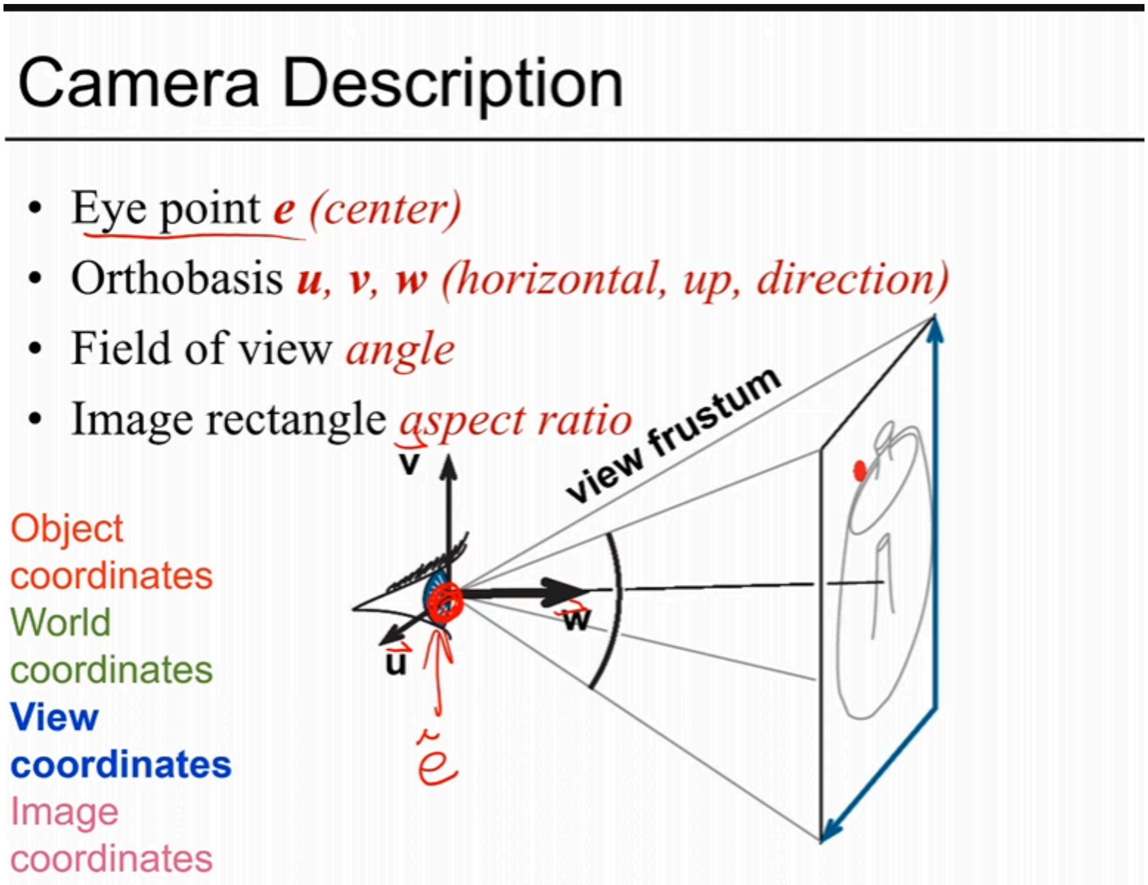
I'd like to find the very first intersection point, t,

- Camera Obscura (Pinhole Camera)



These artists in these 1500s were pretty clever, really.

- Camera Description

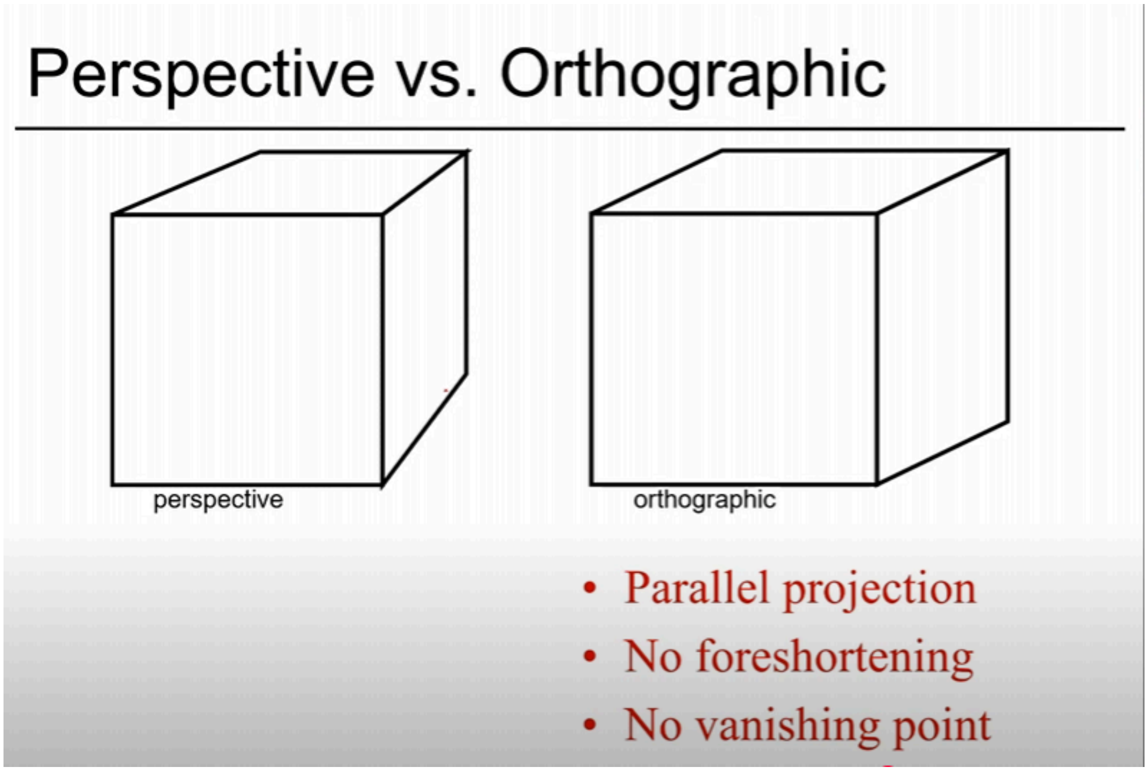


Camera Description

- Eye point *e (center)*
- Orthobasis ***u*, *v*, *w*** *(horizontal, up, direction)*
- Field of view *angle*
- Image rectangle *aspect ratio*

Object coordinates
World coordinates
View coordinates
Image coordinates

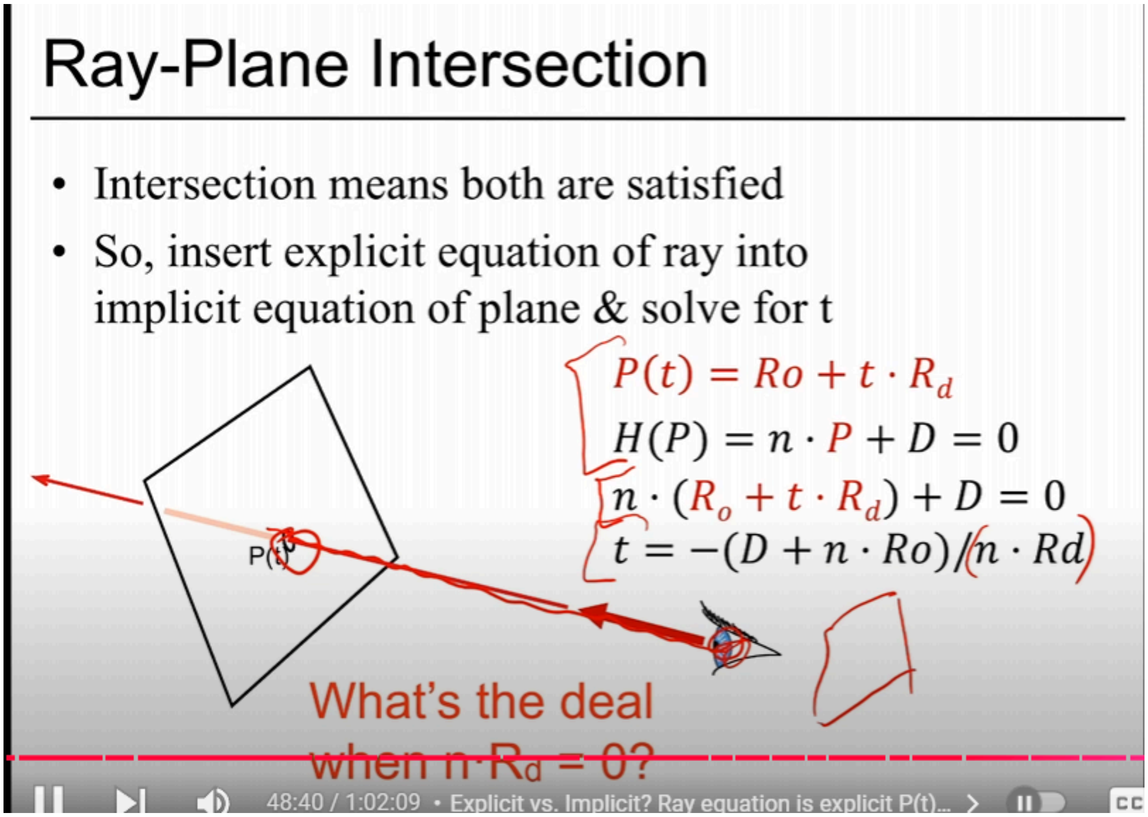
- Image Coordinates
- Perspective vs. Orhographic



Perspective vs. Orthographic

- Parallel projection
- No foreshortening
- No vanishing point

- Ray-Plane Intersection



Ray-Plane Intersection

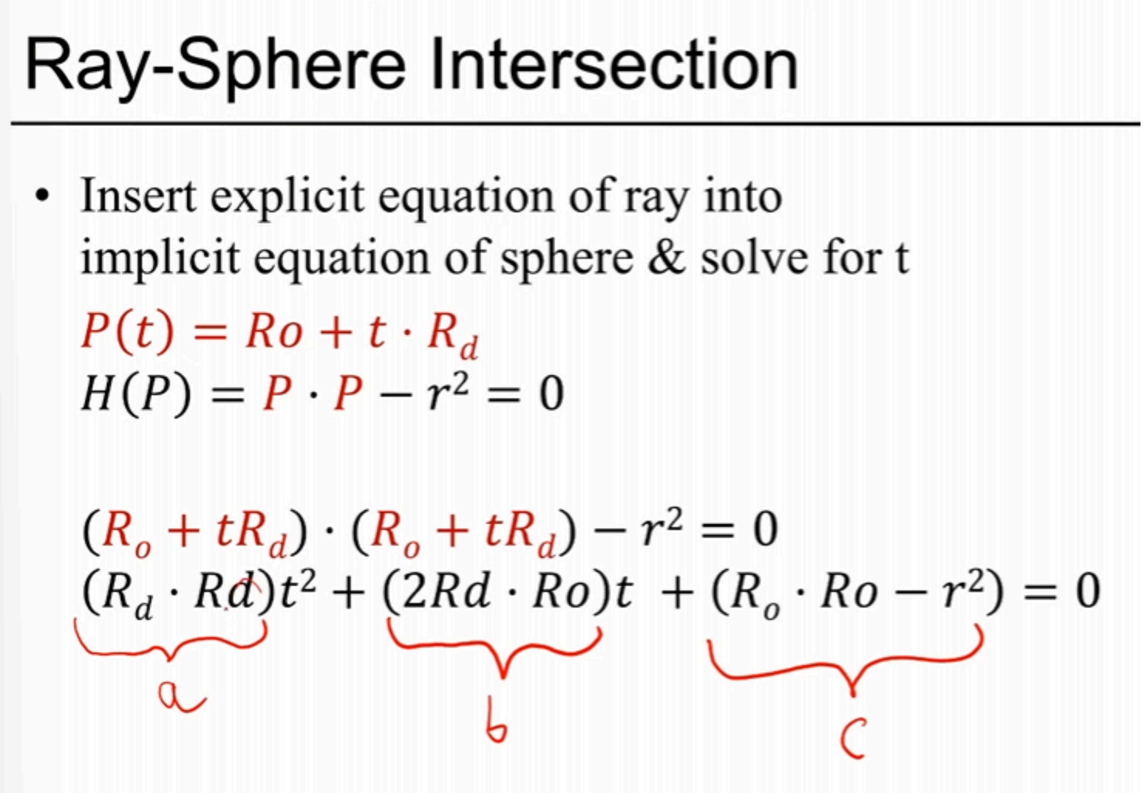
- Intersection means both are satisfied
- So, insert explicit equation of ray into implicit equation of plane & solve for t

$$P(t) = Ro + t\cdot R_d$$
$$H(P) = n\cdot P + D = 0$$
$$n\cdot(R_o + t\cdot R_d) + D = 0$$
$$t = -(D + n\cdot Ro)/(n\cdot Rd)$$

What's the deal when $n\cdot R_d = 0$?

P(t)
48:40 / 1:02:09 • Explicit vs. Implicit? Ray equation is explicit P(t)...
CC

- Ray-Sphere Interrsection

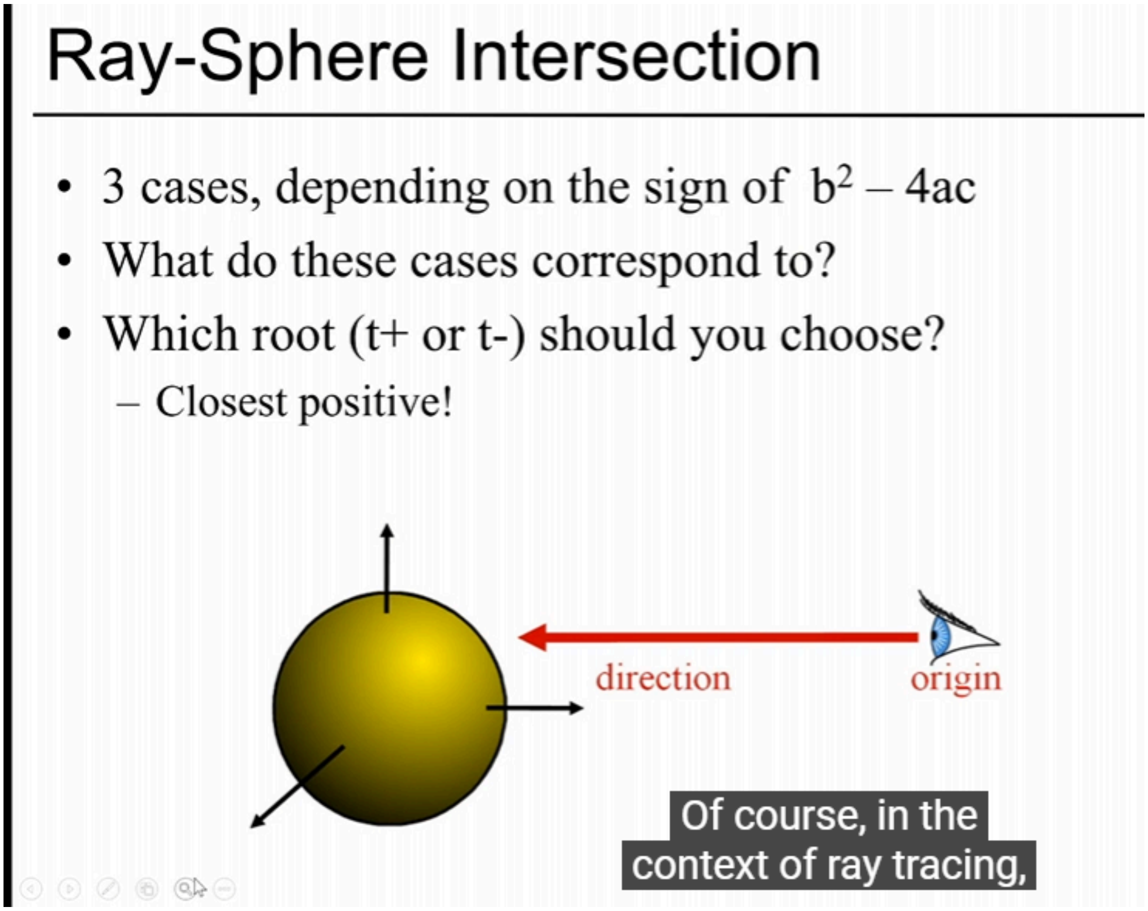


Ray-Sphere Intersection

- Insert explicit equation of ray into implicit equation of sphere & solve for t

$$P(t) = Ro + t\cdot R_d$$
$$H(P) = P\cdot P - r^2 = 0$$

$$(R_o + tR_d)\cdot(R_o + tR_d) - r^2 = 0$$
$$\underbrace{(R_d\cdot Rd)}_{a}t^2 + \underbrace{(2Rd\cdot Ro)}_{b}t + \underbrace{(R_o\cdot Ro - r^2)}_{c} = 0$$



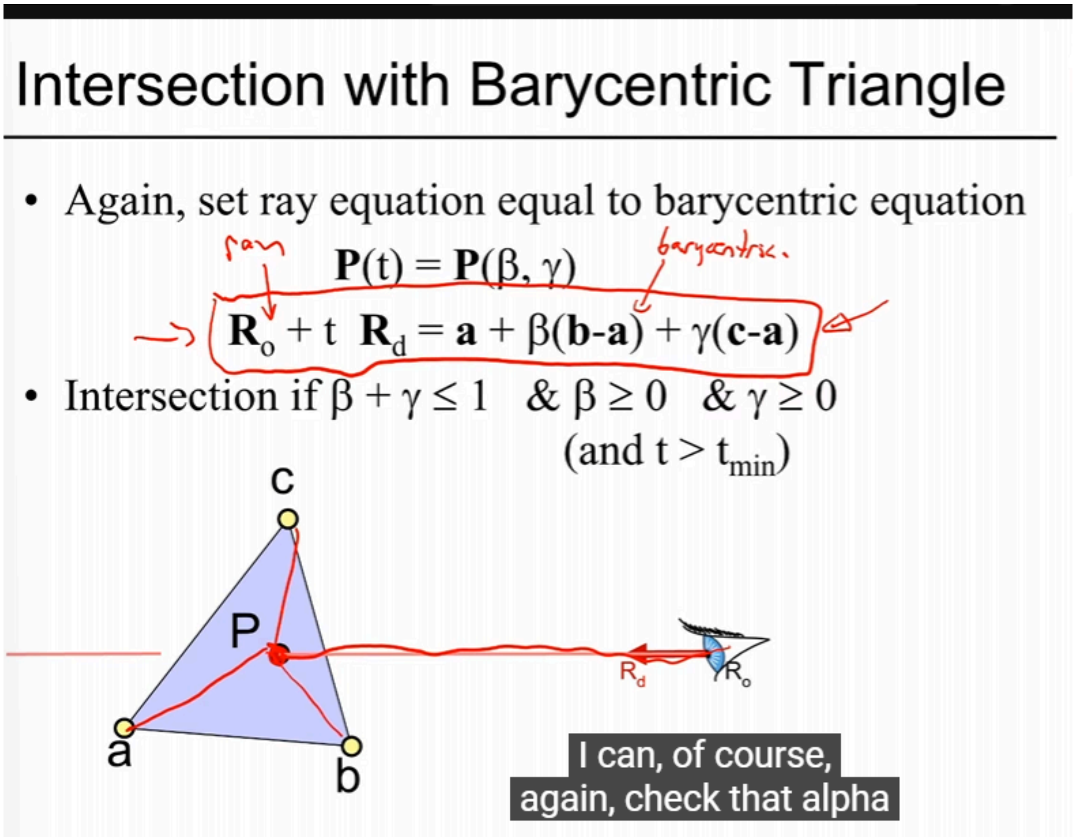
Ray-Sphere Intersection

- 3 cases, depending on the sign of $b^2 - 4ac$
- What do these cases correspond to?
- Which root (t+ or t-) should you choose?
  - Closest positive!

Of course, in the context of ray tracing,

- L10: Ray Casting II
  - Ray Casting
    - Ray-Triangle Intersection
      - Barycentric coorinates

- Intersection with Barycentric Triangle



Intersection with Barycentric Triangle

- Again, set ray equation equal to barycentric equation

ray barycentric.

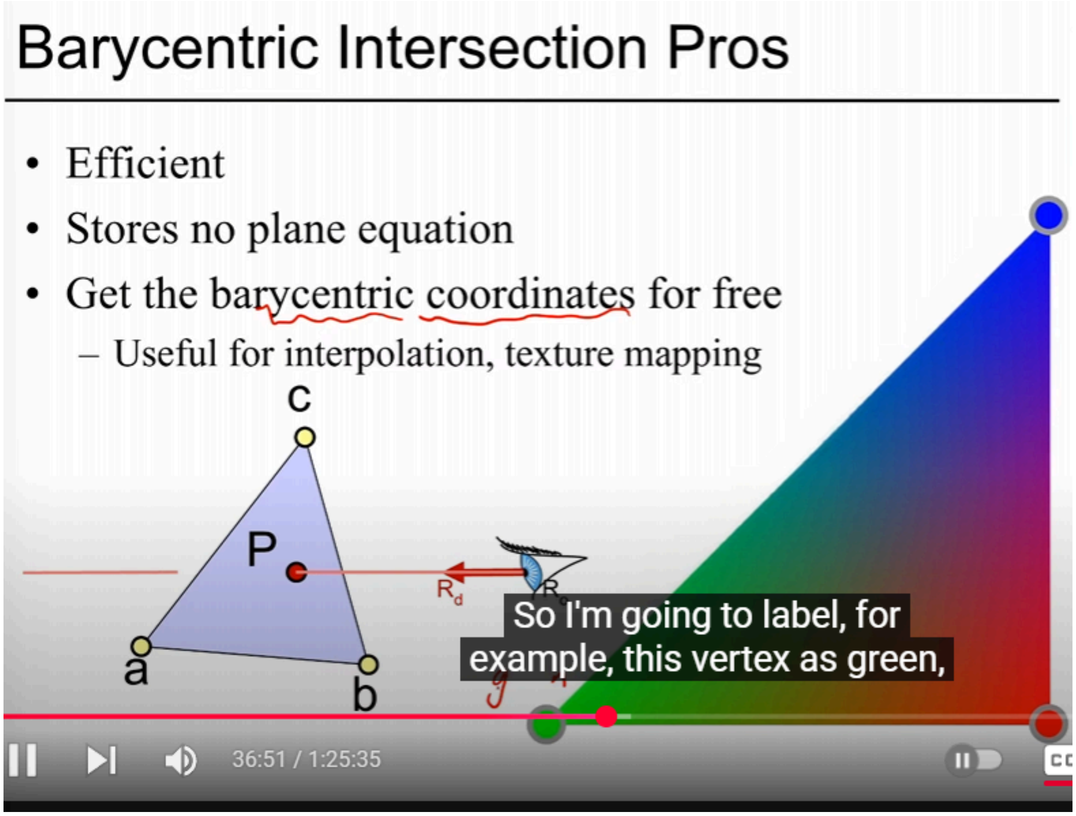
$$\mathbf{P}(t) = \mathbf{P}(\beta, \gamma)$$

$$\boxed{\mathbf{R}_o + t\ \mathbf{R}_d = \mathbf{a} + \beta(\mathbf{b}\text{-}\mathbf{a}) + \gamma(\mathbf{c}\text{-}\mathbf{a})}$$

- Intersection if $\beta + \gamma \leq 1$ & $\beta \geq 0$ & $\gamma \geq 0$ (and $t > t_{min}$)

I can, of course, again, check that alpha

- Barycentric Intersection Pros

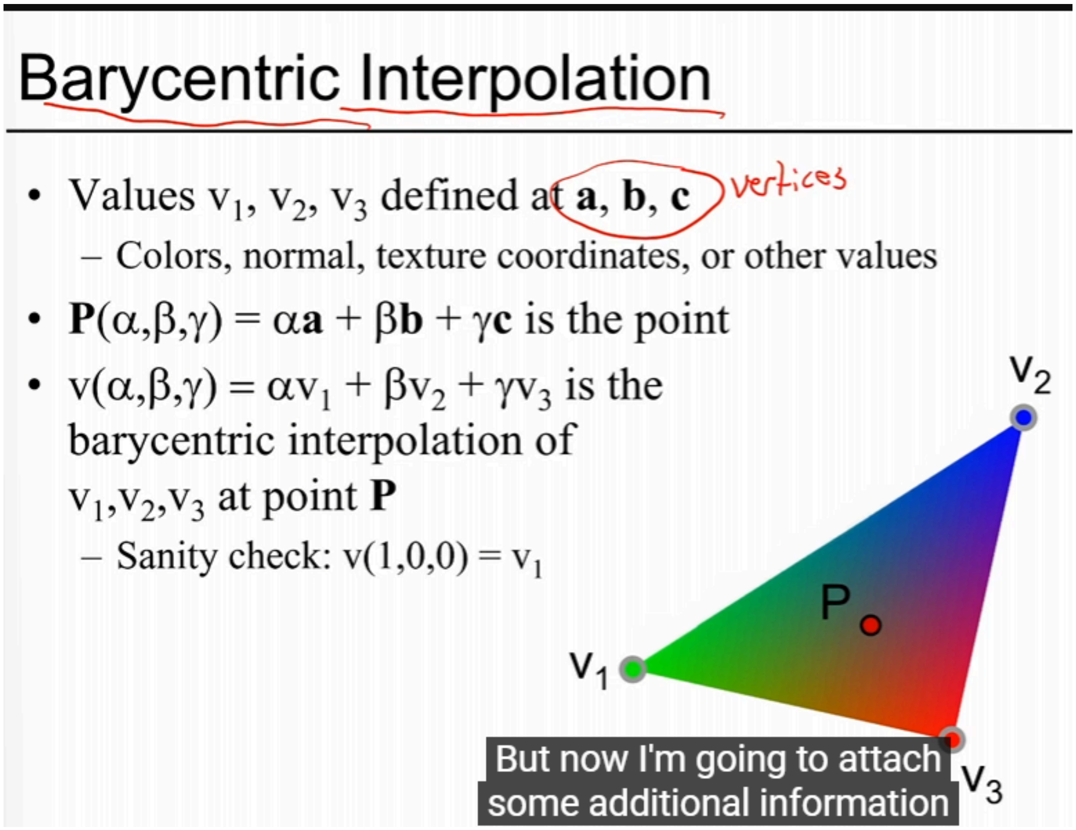


Barycentric Intersection Pros

- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
  - Useful for interpolation, texture mapping

So I'm going to label, for example, this vertex as green,

- Barycentric Interpolation

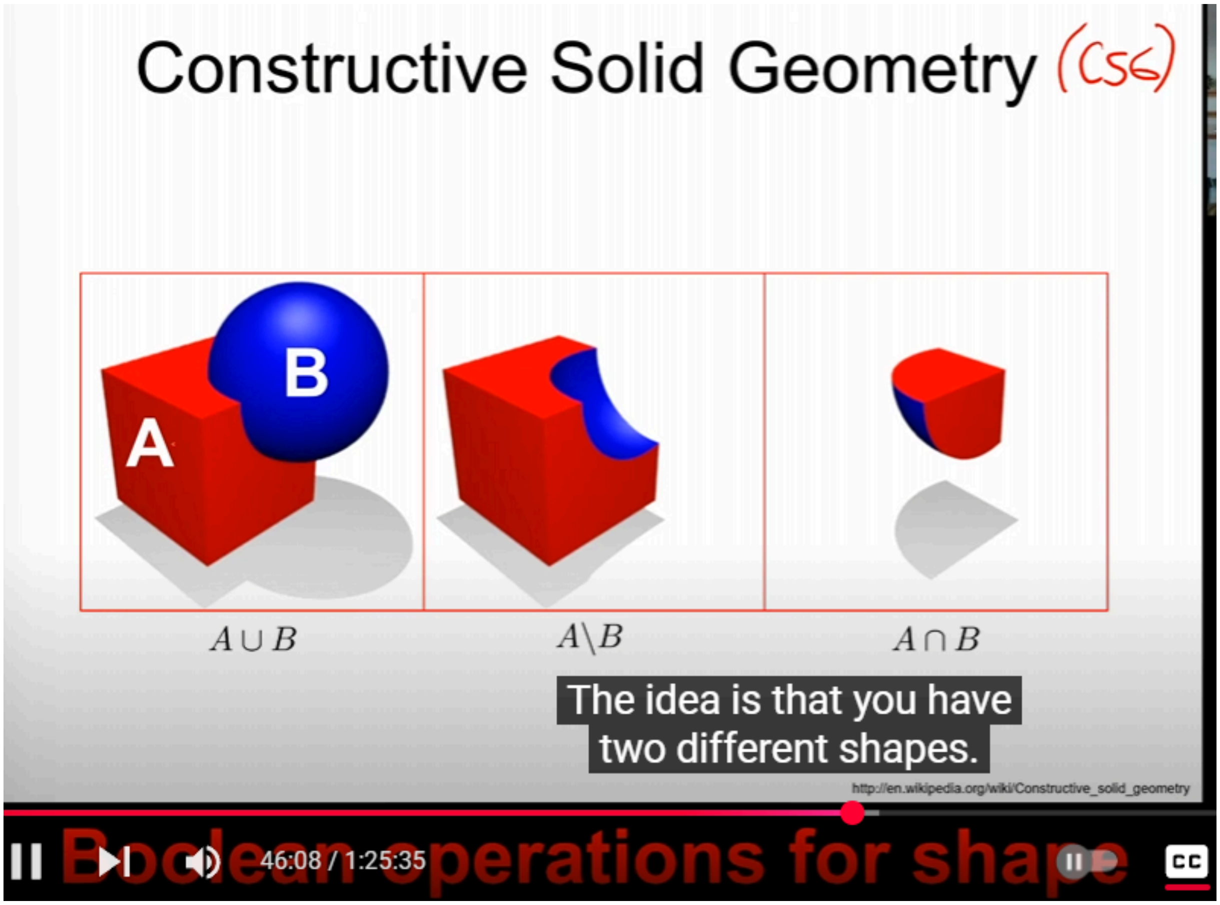


Barycentric Interpolation

- Values v_1, v_2, v_3 defined at **a**, **b**, **c** vertices
  - Colors, normal, texture coordinates, or other values
- $\mathbf{P}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$ is the point
- $v(\alpha,\beta,\gamma) = \alpha v_1 + \beta v_2 + \gamma v_3$ is the barycentric interpolation of v_1,v_2,v_3 at point **P**
  - Sanity check: $v(1,0,0) = v_1$

But now I'm going to attach some additional information

- Constructive Solid Geometry (CSG)



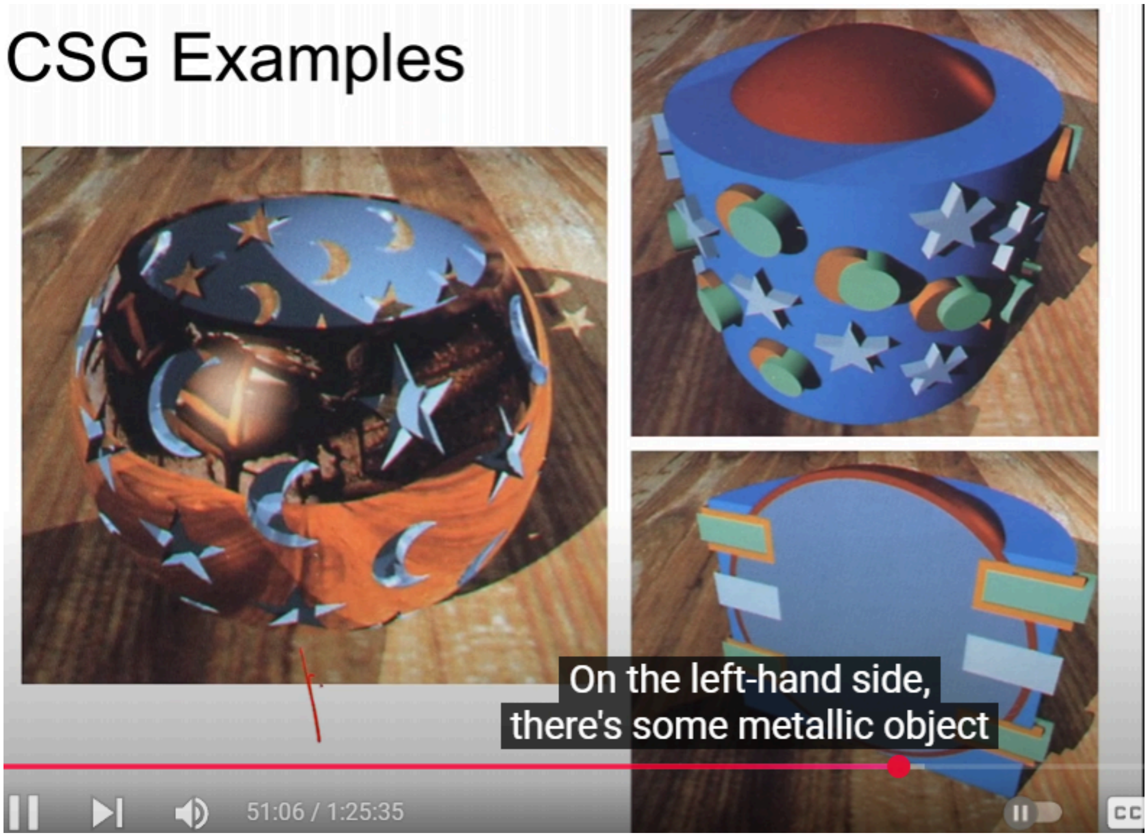
Constructive Solid Geometry (CSG)

The idea is that you have two different shapes.

http://en.wikipedia.org/wiki/Constructive_solid_geometry

Boolean operations for shape

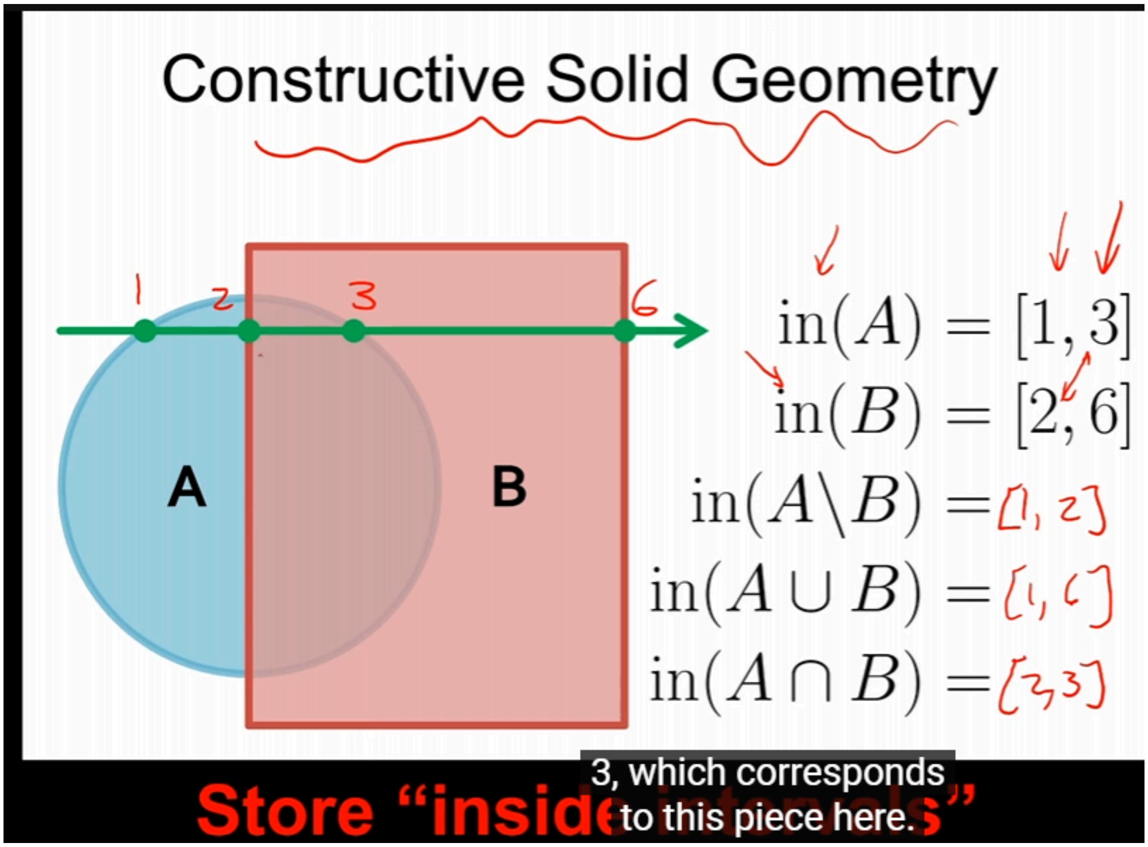
- Example



CSG Examples

On the left-hand side,
there's some metallic object

- Ray Tracing CSG



Constructive Solid Geometry

$$\mathrm{in}(A) = [1,3]$$

$$\mathrm{in}(B) = [2,6]$$

$$\mathrm{in}(A\backslash B) = [1,2]$$

$$\mathrm{in}(A \cup B) = [1,6]$$

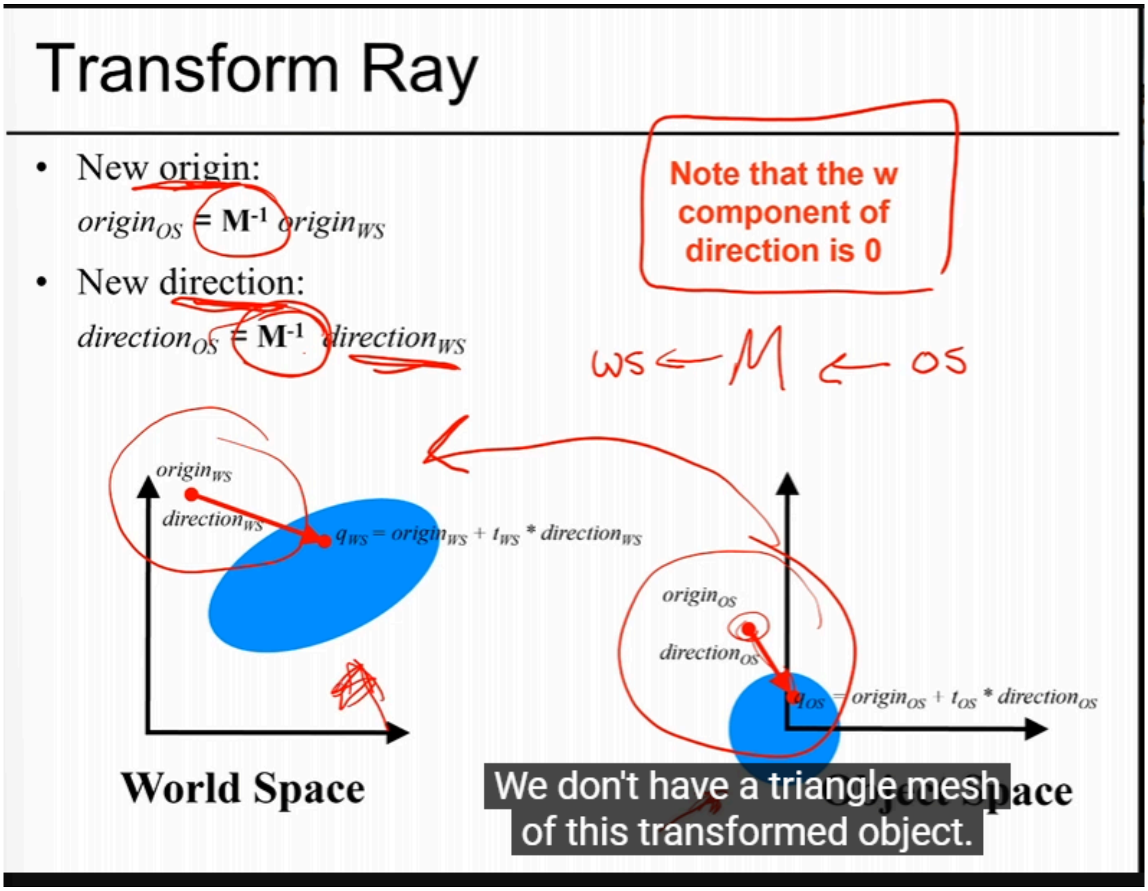
$$\mathrm{in}(A \cap B) = [2,3]$$

Store “inside intervals”

3, which corresponds
to this piece here.

- Instancing and Transformations

- Transform Ray



Transform Ray

- New origin:

  $origin_{OS} = \mathbf{M}^{-1}\, origin_{WS}$

- New direction:

  $direction_{OS} = \mathbf{M}^{-1}\, direction_{WS}$

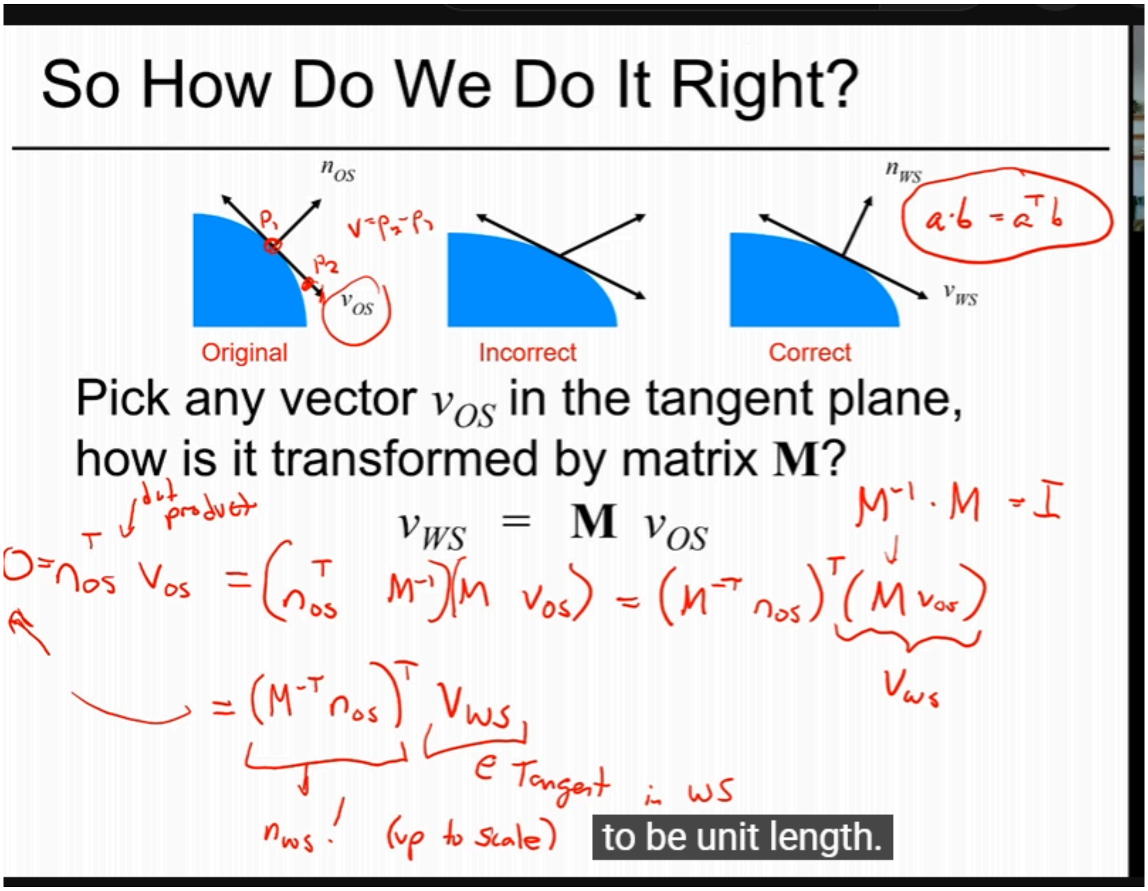
Note that the w component of direction is 0

ws ← M ← os

$origin_{WS}$
$direction_{WS}$
$q_{WS} = origin_{WS} + t_{WS} * direction_{WS}$
World Space
$origin_{OS}$
$direction_{OS}$
$q_{OS} = origin_{OS} + t_{OS} * direction_{OS}$
Object Space

We don't have a triangle mesh of this transformed object.

- Calculated Normal after transformed



So How Do We Do It Right?

Pick any vector v_{OS} in the tangent plane, how is it transformed by matrix $\mathbf{M}$?

$$v_{WS} = \mathbf{M}\, v_{OS}$$

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

dot product

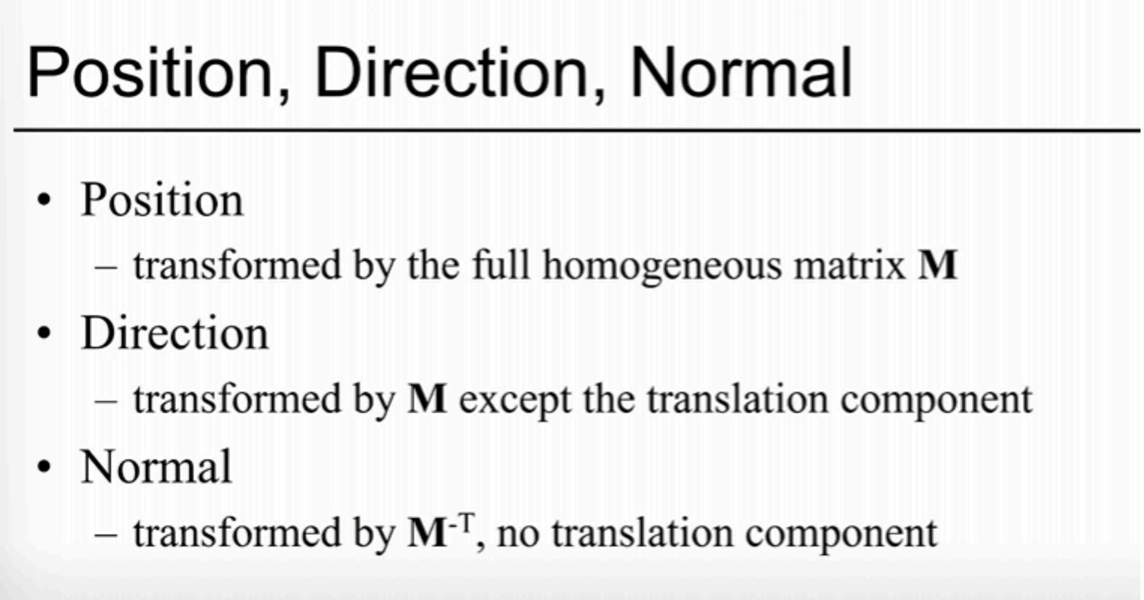
$$0 = n_{os}^T v_{os} = (n_{os}^T M^{-1})(M v_{os}) = (M^{-T} n_{os})^T \underbrace{(M v_{os})}_{v_{ws}}$$

$$= \underbrace{(M^{-T} n_{os})^T}_{n_{ws}!} \underbrace{v_{ws}}_{\in \text{Tangent in WS}}$$

(up to scale)

to be unit length.

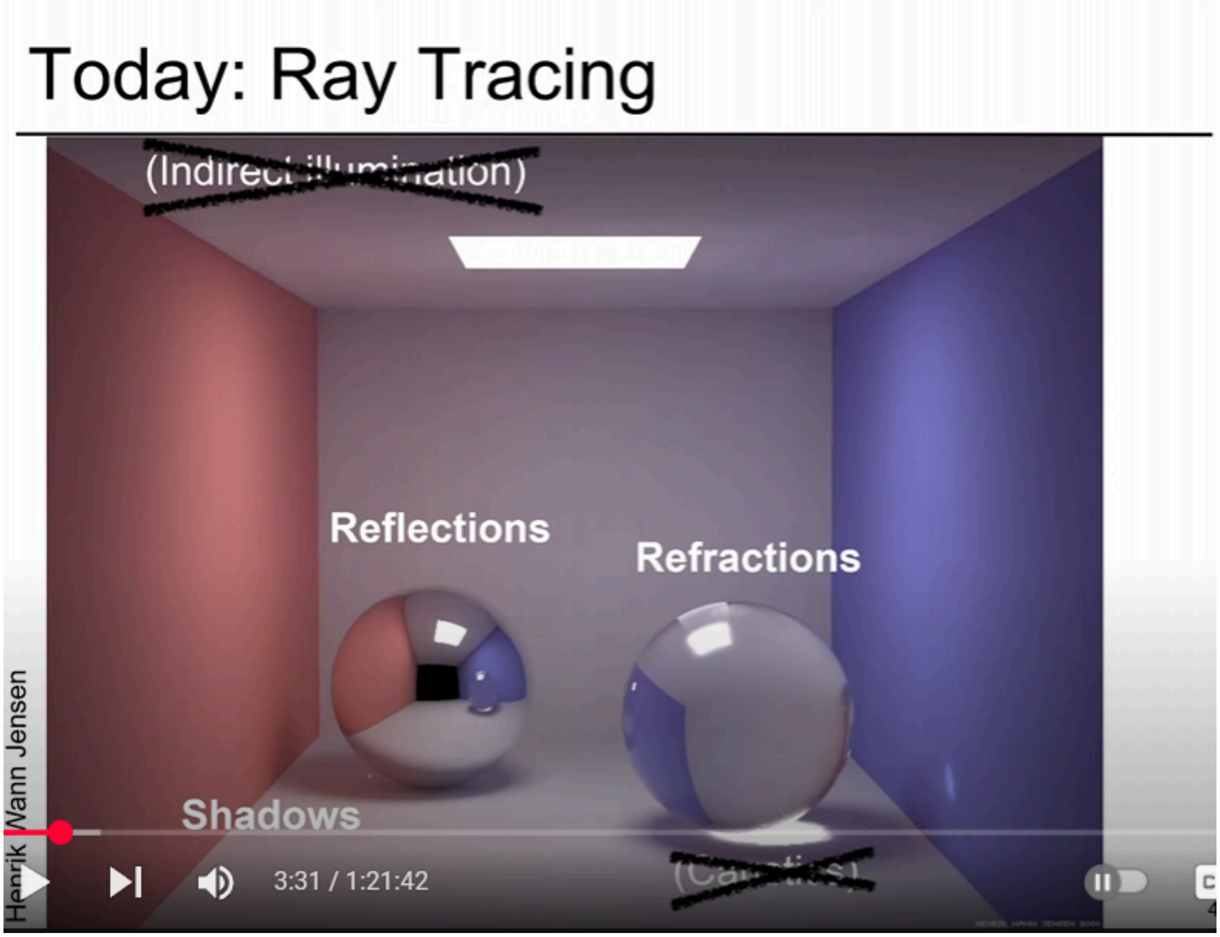
    - Position, Direction, Normal



Position, Direction, Normal

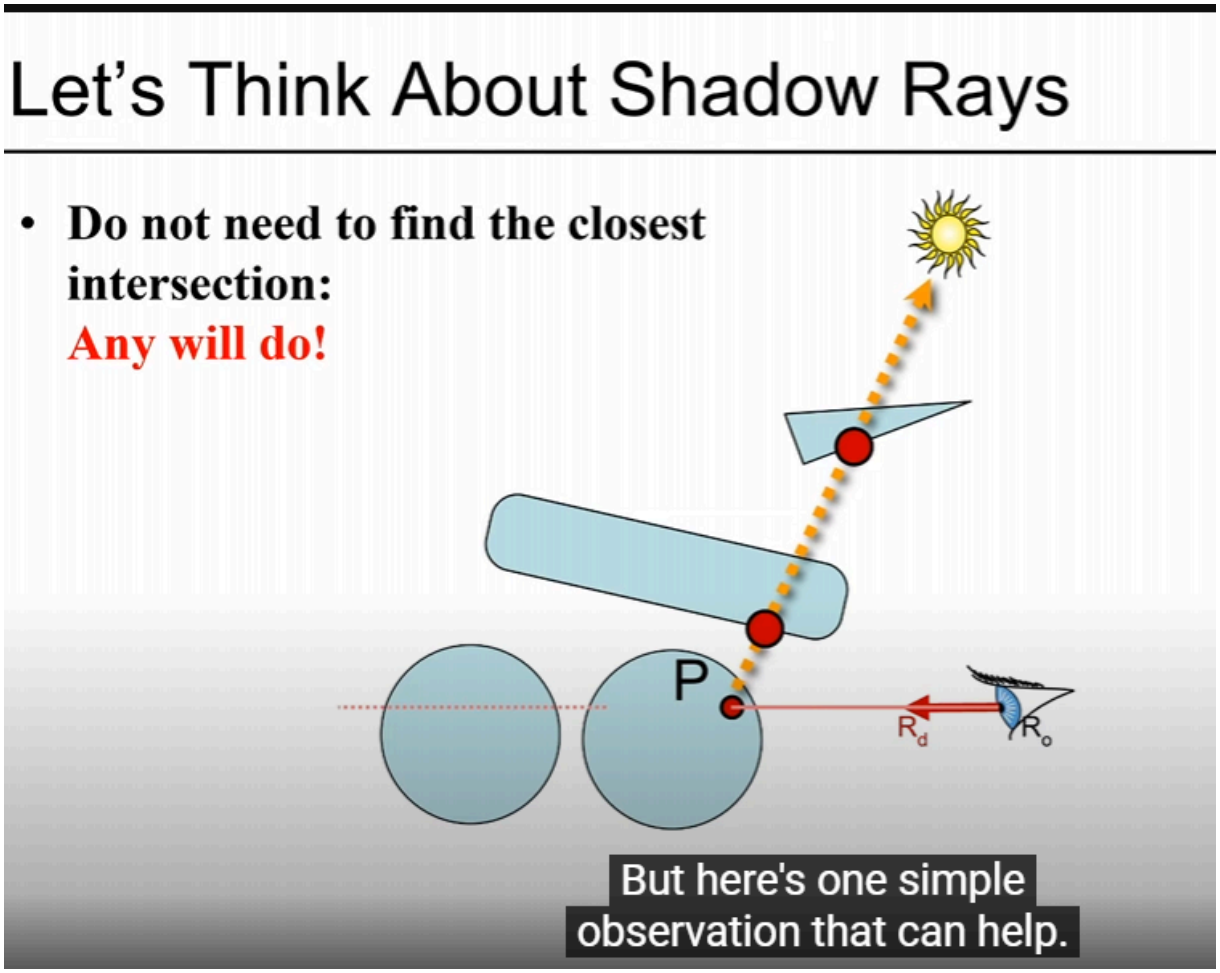
- Position
  - transformed by the full homogeneous matrix **M**
- Direction
  - transformed by **M** except the translation component
- Normal
  - transformed by $\mathbf{M}^{-T}$, no translation component

- **L11: Ray Tracing**
  - Example



Today: Ray Tracing

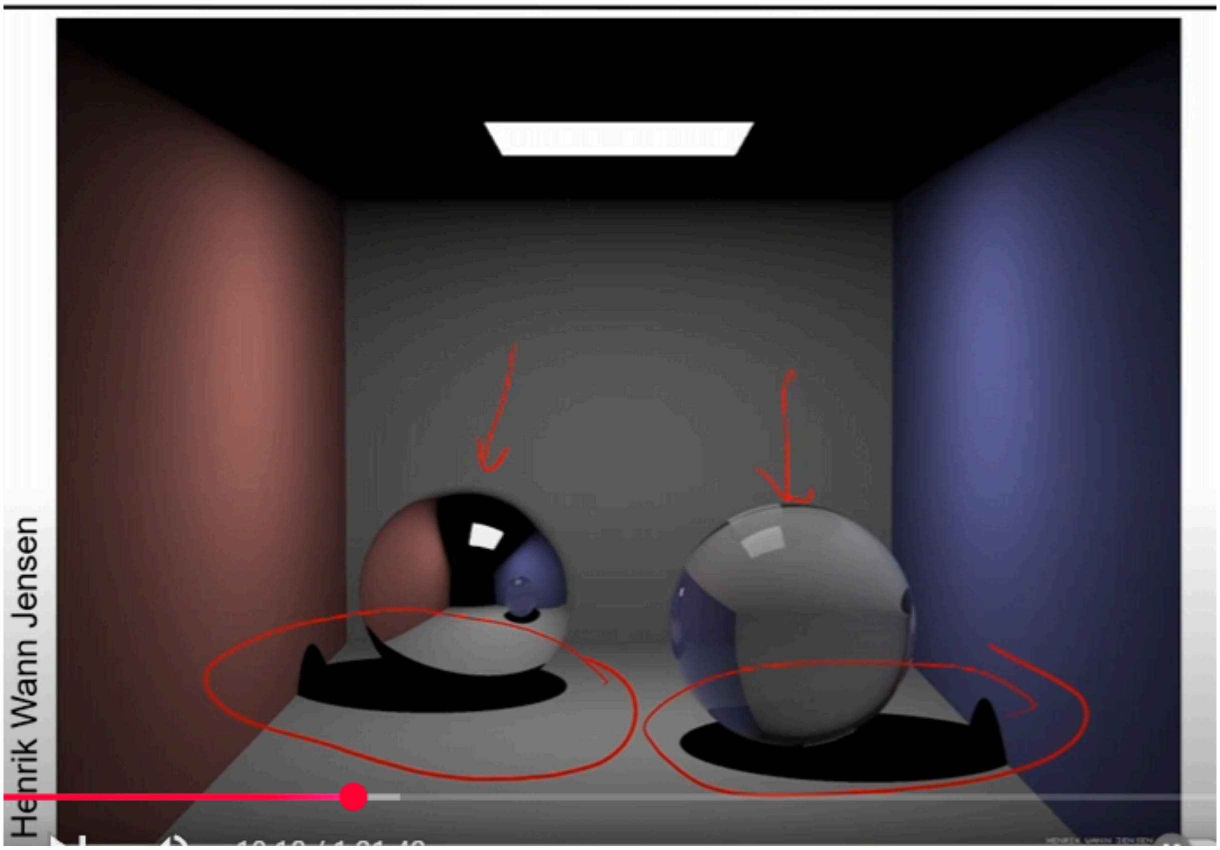
- Shadows



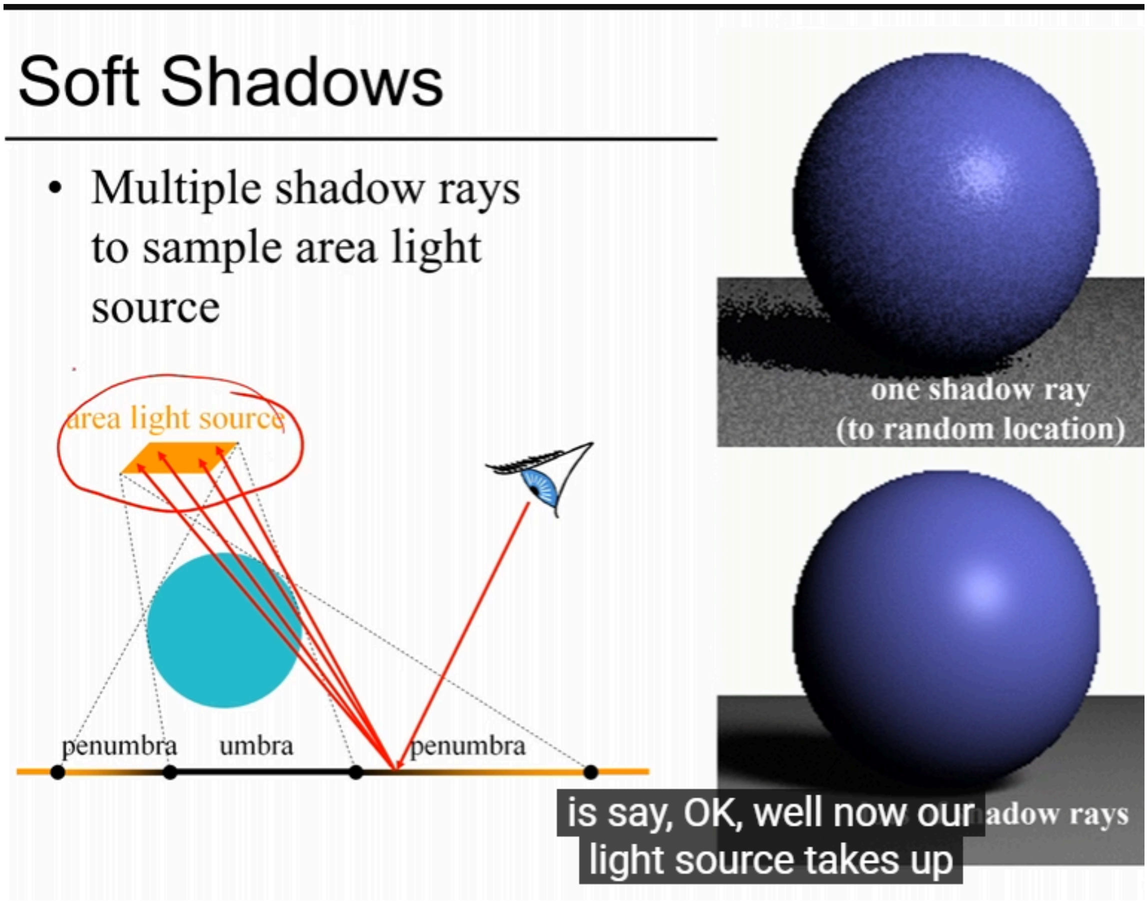
Let’s Think About Shadow Rays

- **Do not need to find the closest intersection:**
  Any will do!

But here's one simple observation that can help.



- Soft Shadow

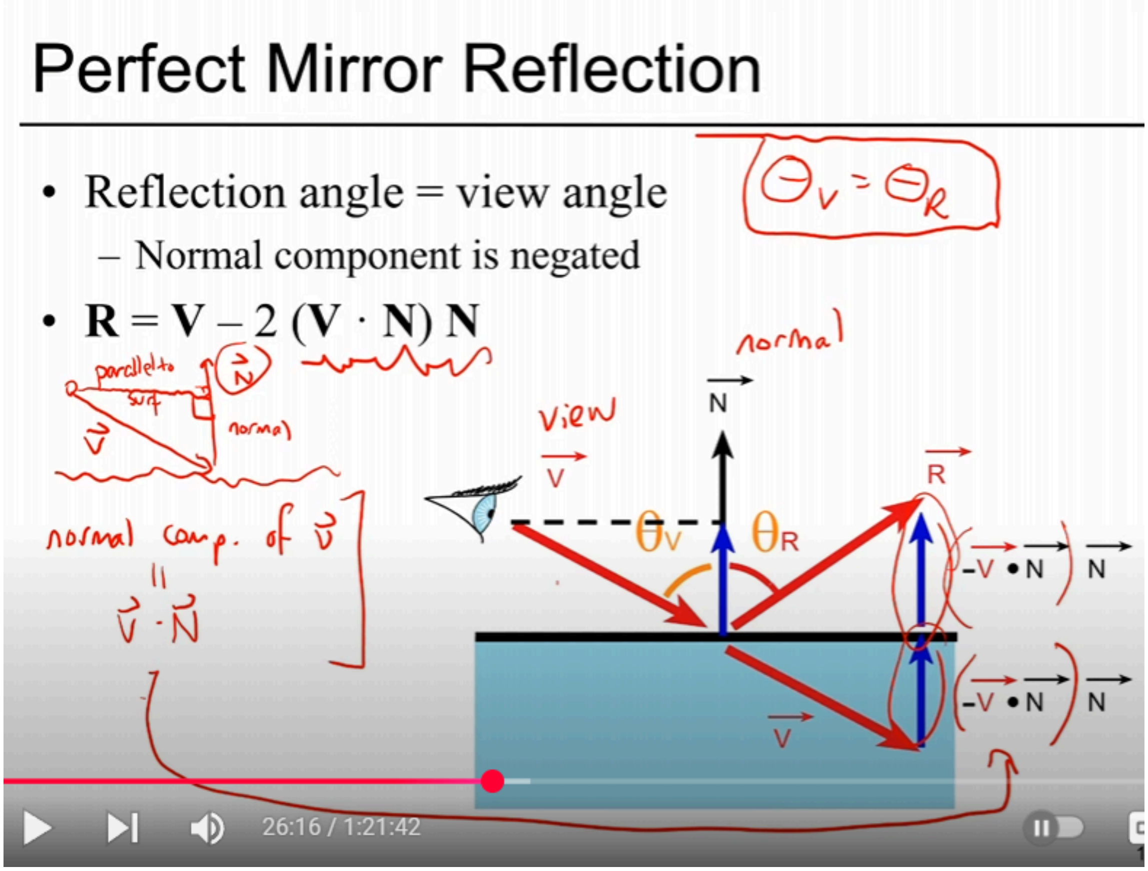


Soft Shadows

- Multiple shadow rays to sample area light source

is say, OK, well now our
light source takes up

- Reflection
  - Perfect Mirror Relfection



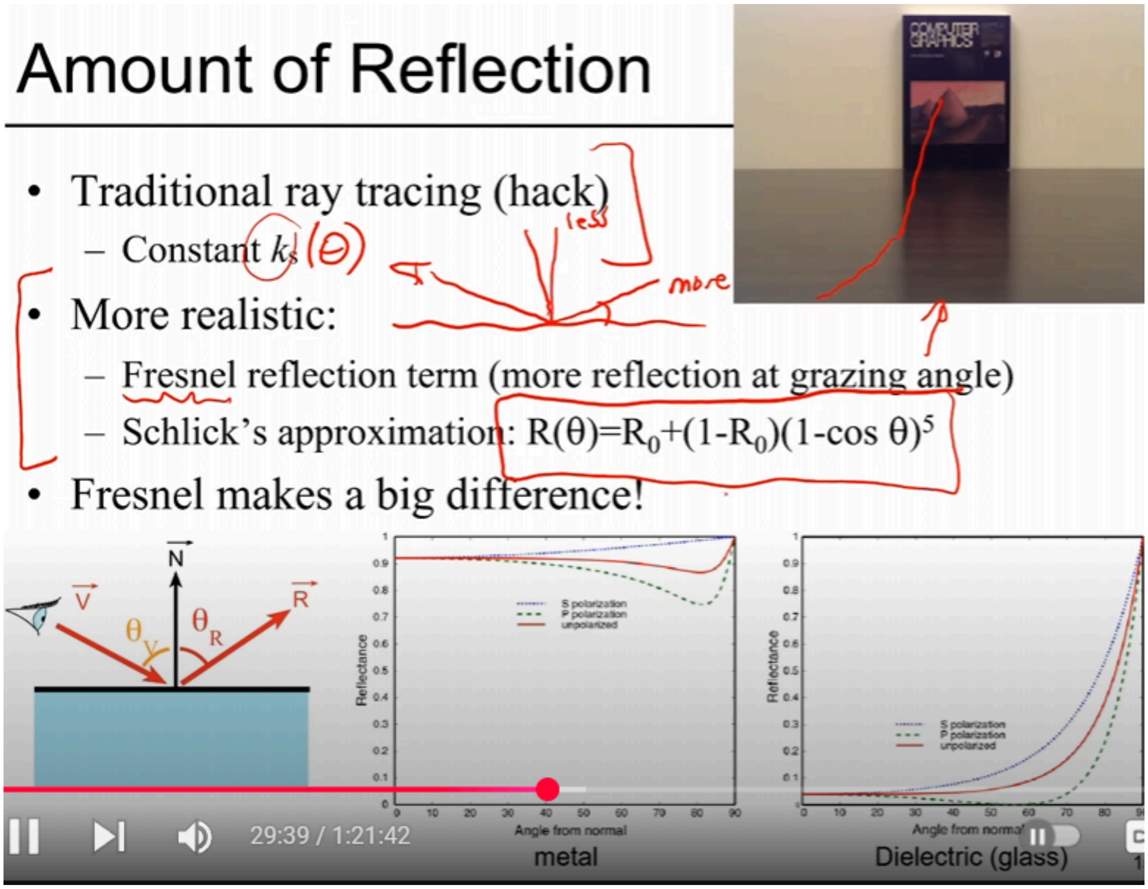
Perfect Mirror Reflection

- Reflection angle = view angle
  - Normal component is negated
- $\mathbf{R} = \mathbf{V} - 2\,(\mathbf{V} \cdot \mathbf{N})\,\mathbf{N}$

$\boxed{\theta_V = \theta_R}$

normal comp. of $\vec{V}$ $= \vec{V} \cdot \vec{N}$

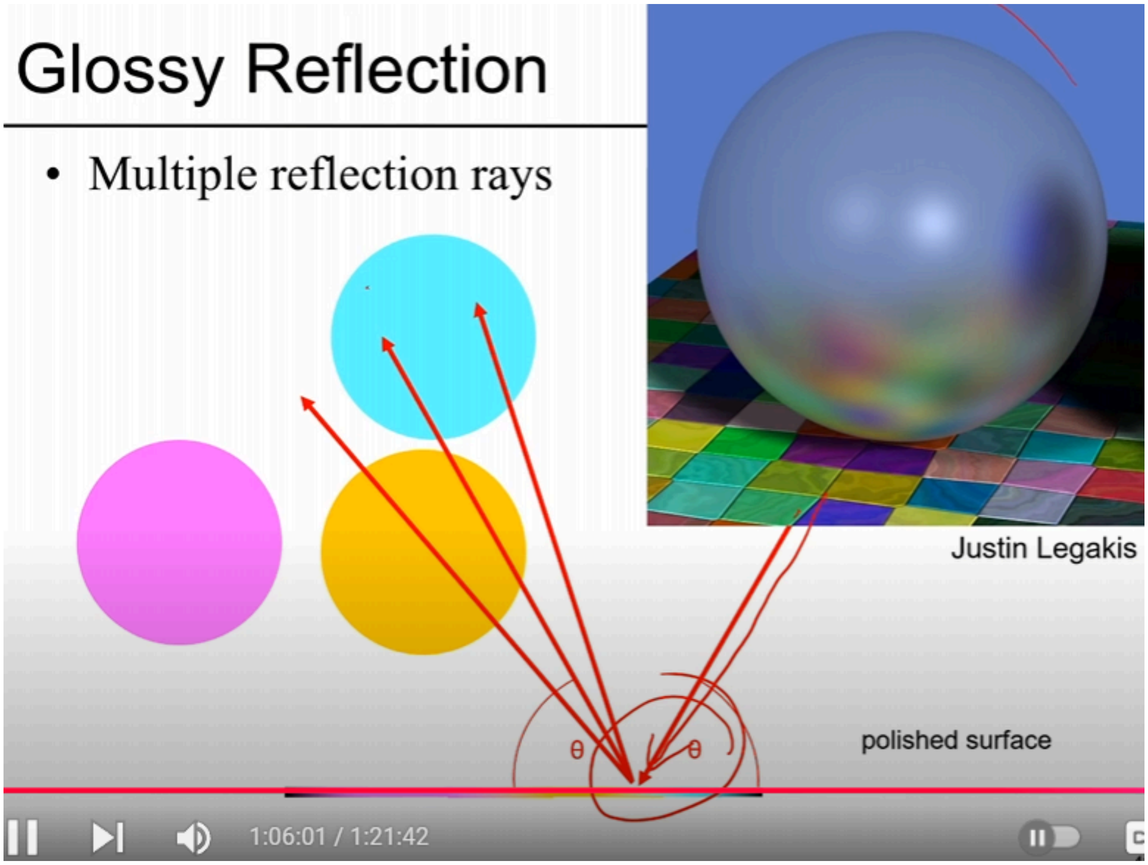
- Amount of Relrection



Amount of Reflection

- Traditional ray tracing (hack)
  - Constant k_s (θ)
- More realistic:
  - Fresnel reflection term (more reflection at grazing angle)
  - Schlick's approximation: $R(\theta)=R_0+(1-R_0)(1-\cos\theta)^5$
- Fresnel makes a big difference!

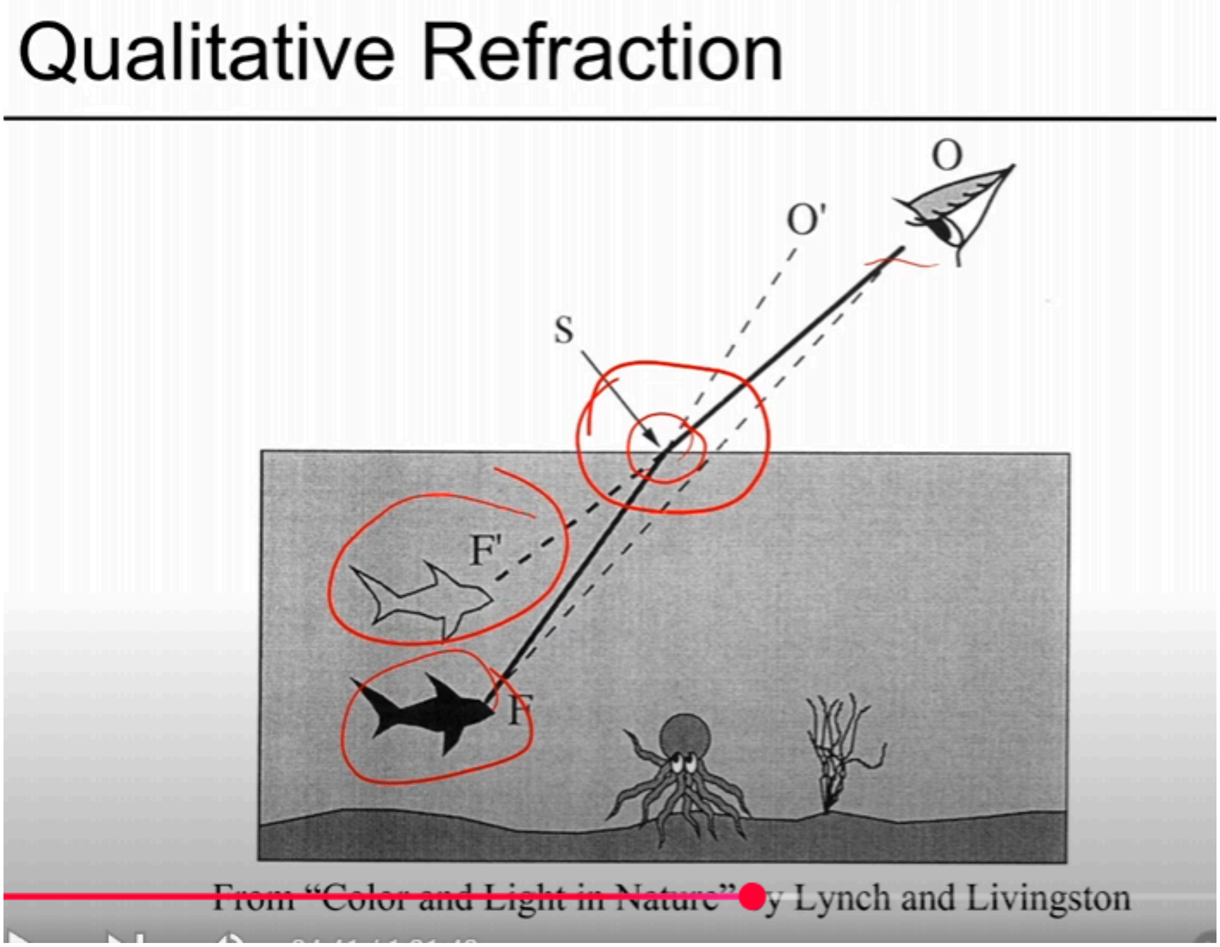
- Glossy Refection



Glossy Reflection

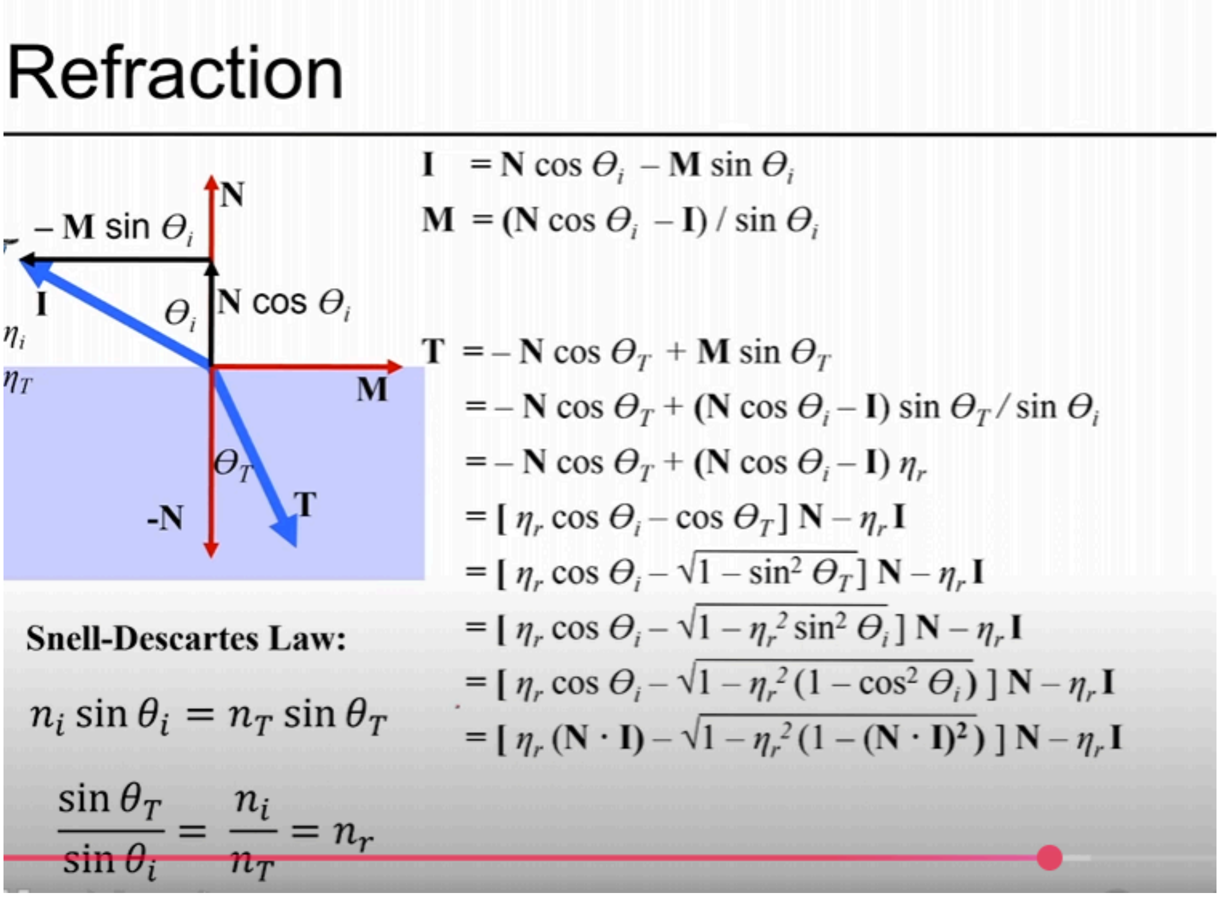
- Multiple reflection rays

- Refraction



Qualitative Refraction

From “Color and Light in Nature” by Lynch and Livingston



Refraction

$$\mathbf{I} = \mathbf{N}\cos\Theta_i - \mathbf{M}\sin\Theta_i$$

$$\mathbf{M} = (\mathbf{N}\cos\Theta_i - \mathbf{I}) / \sin\Theta_i$$

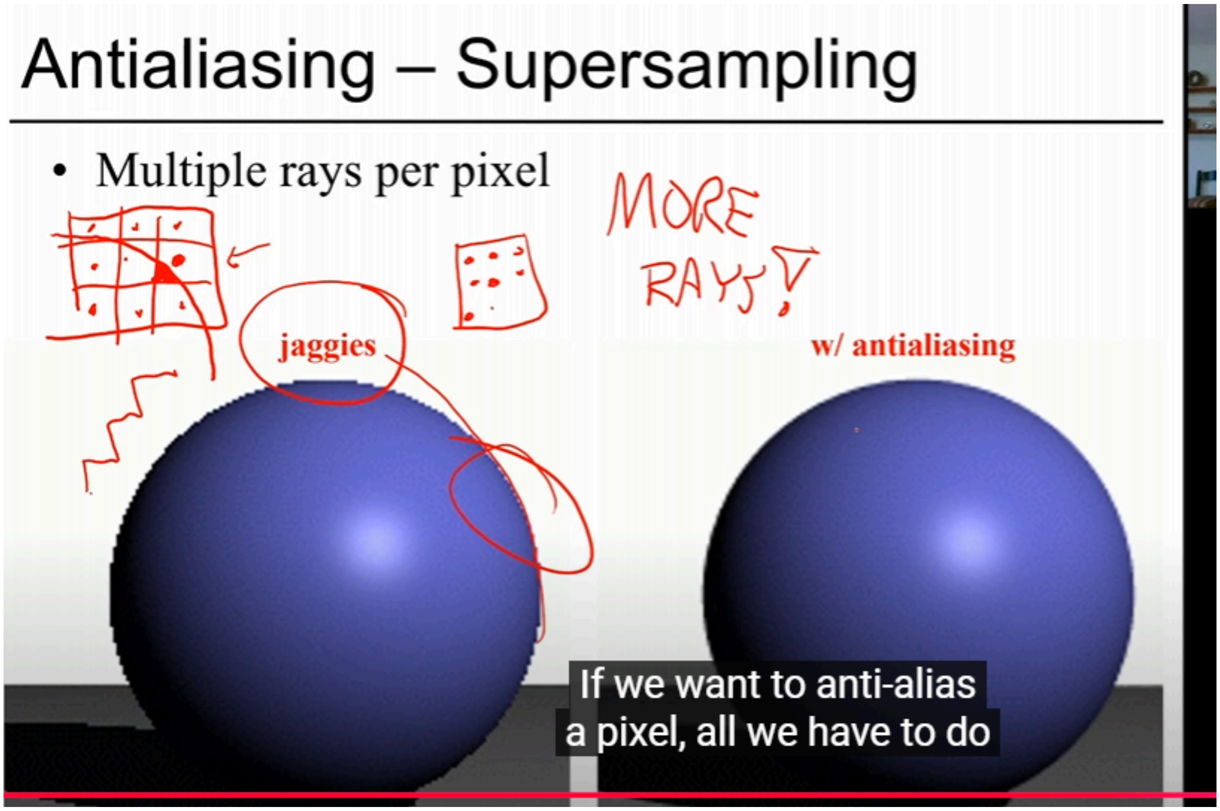
$$\begin{aligned}
\mathbf{T} &= -\mathbf{N}\cos\Theta_T + \mathbf{M}\sin\Theta_T \\
&= -\mathbf{N}\cos\Theta_T + (\mathbf{N}\cos\Theta_i - \mathbf{I})\sin\Theta_T / \sin\Theta_i \\
&= -\mathbf{N}\cos\Theta_T + (\mathbf{N}\cos\Theta_i - \mathbf{I})\,\eta_r \\
&= [\,\eta_r\cos\Theta_i - \cos\Theta_T\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r\cos\Theta_i - \sqrt{1-\sin^2\Theta_T}\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r\cos\Theta_i - \sqrt{1-\eta_r^2\sin^2\Theta_i}\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r\cos\Theta_i - \sqrt{1-\eta_r^2(1-\cos^2\Theta_i)}\,]\,\mathbf{N} - \eta_r\mathbf{I} \\
&= [\,\eta_r(\mathbf{N}\cdot\mathbf{I}) - \sqrt{1-\eta_r^2(1-(\mathbf{N}\cdot\mathbf{I})^2)}\,]\,\mathbf{N} - \eta_r\mathbf{I}
\end{aligned}$$

Snell-Descartes Law:

$$n_i \sin\theta_i = n_T \sin\theta_T$$

$$\frac{\sin\theta_T}{\sin\theta_i} = \frac{n_i}{n_T} = n_r$$

- Antialiasing - Supersampleing



Antialiasing – Supersampling

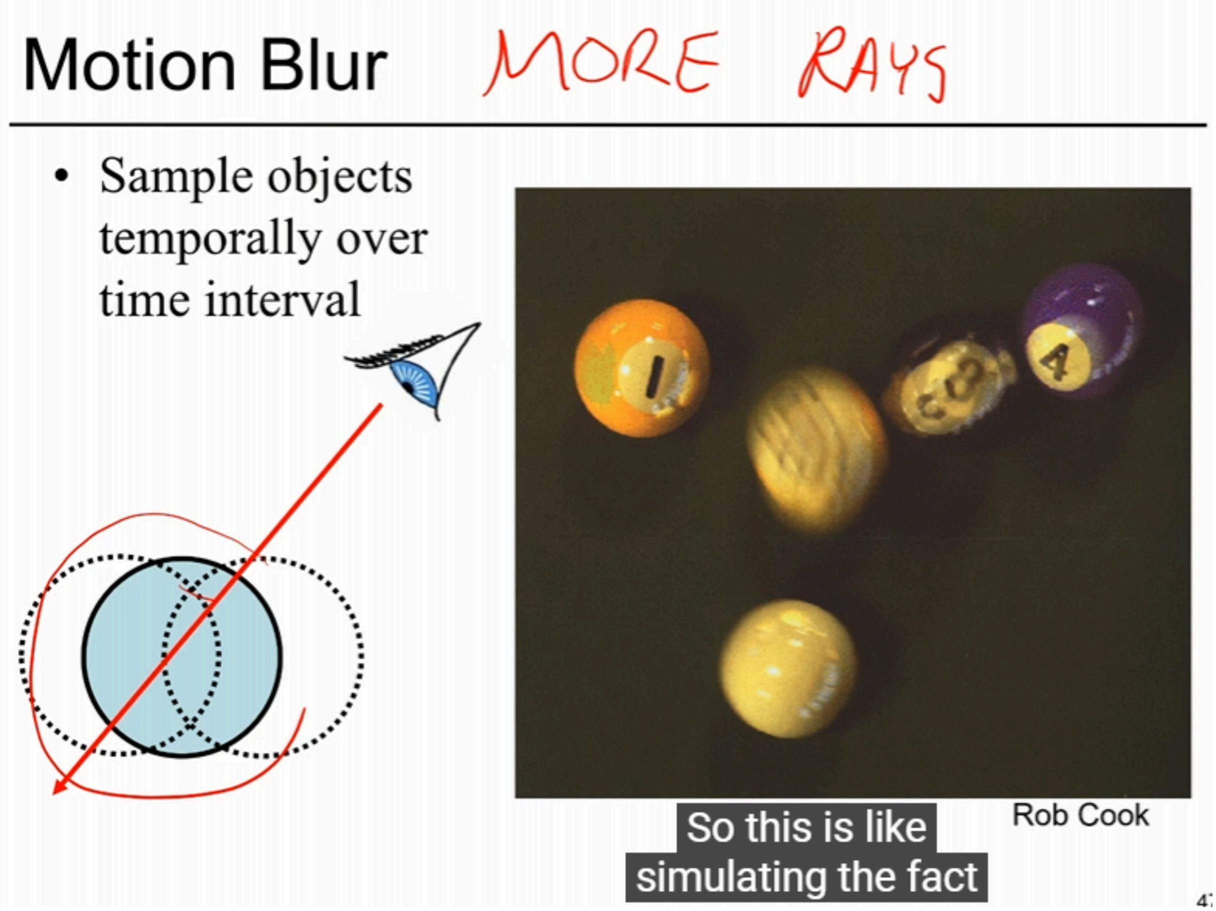
- Multiple rays per pixel

MORE RAYS!

If we want to anti-alias
a pixel, all we have to do

  - Send more ray in the pixel, and average them

- Motion Blur

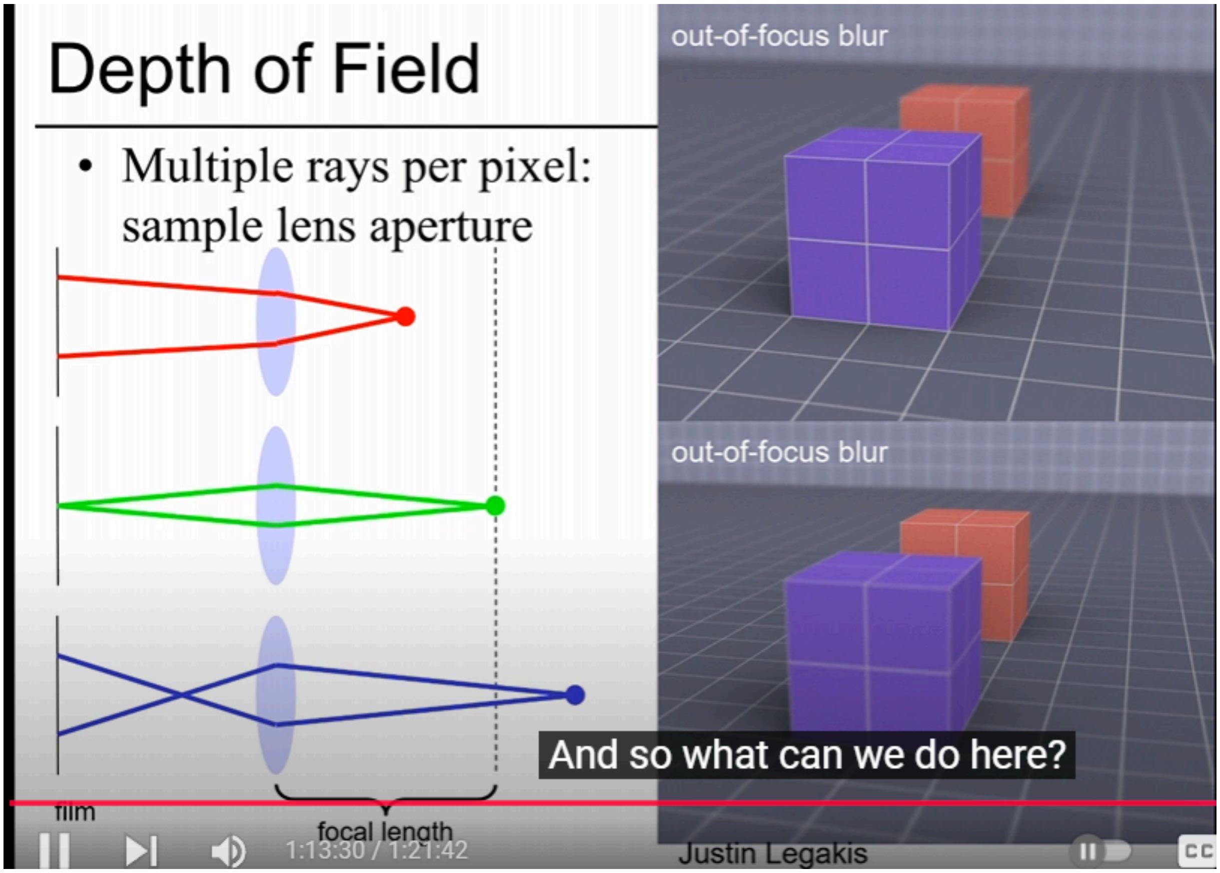


Motion Blur MORE RAYS

- Sample objects temporally over time interval

So this is like
simulating the fact

- Depth of Field



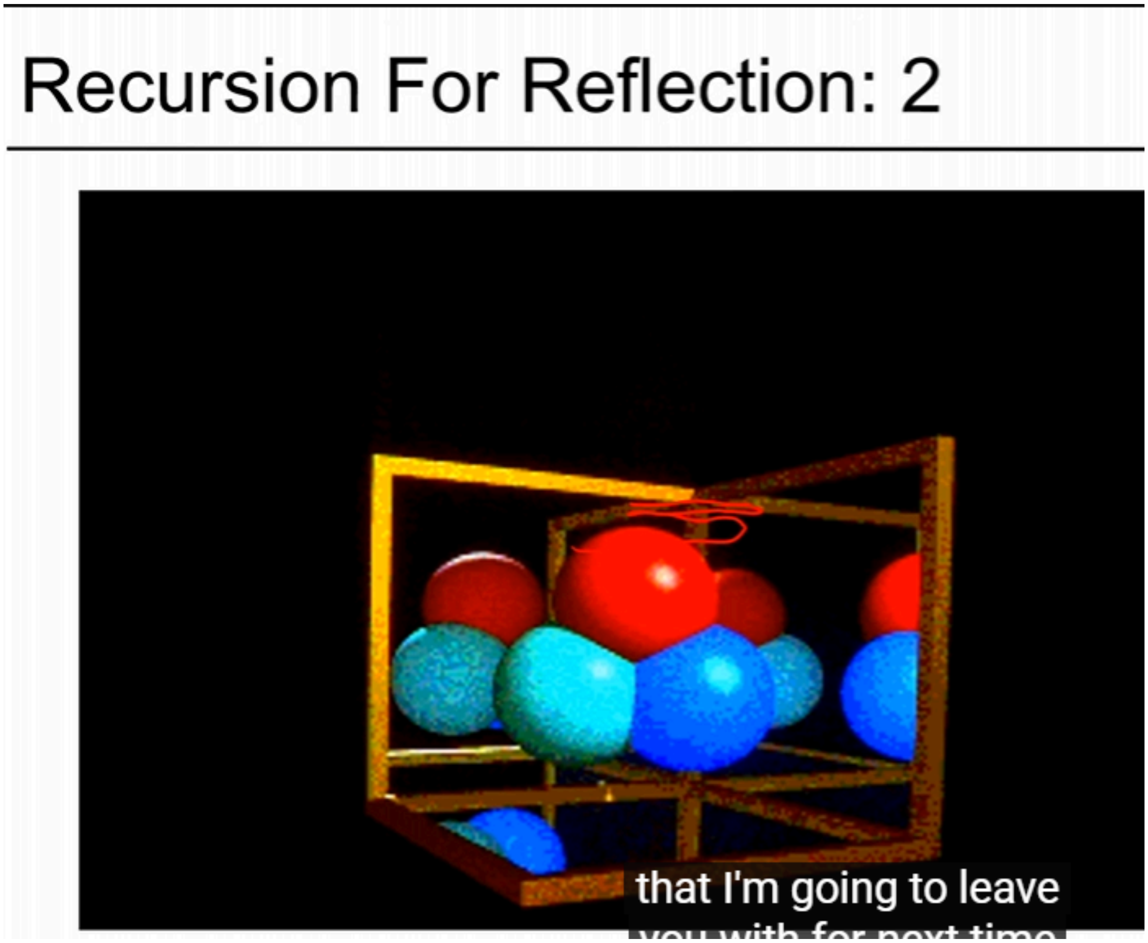
Depth of Field

- Multiple rays per pixel: sample lens aperture

And so what can we do here?

- Recursive Ray Tracing

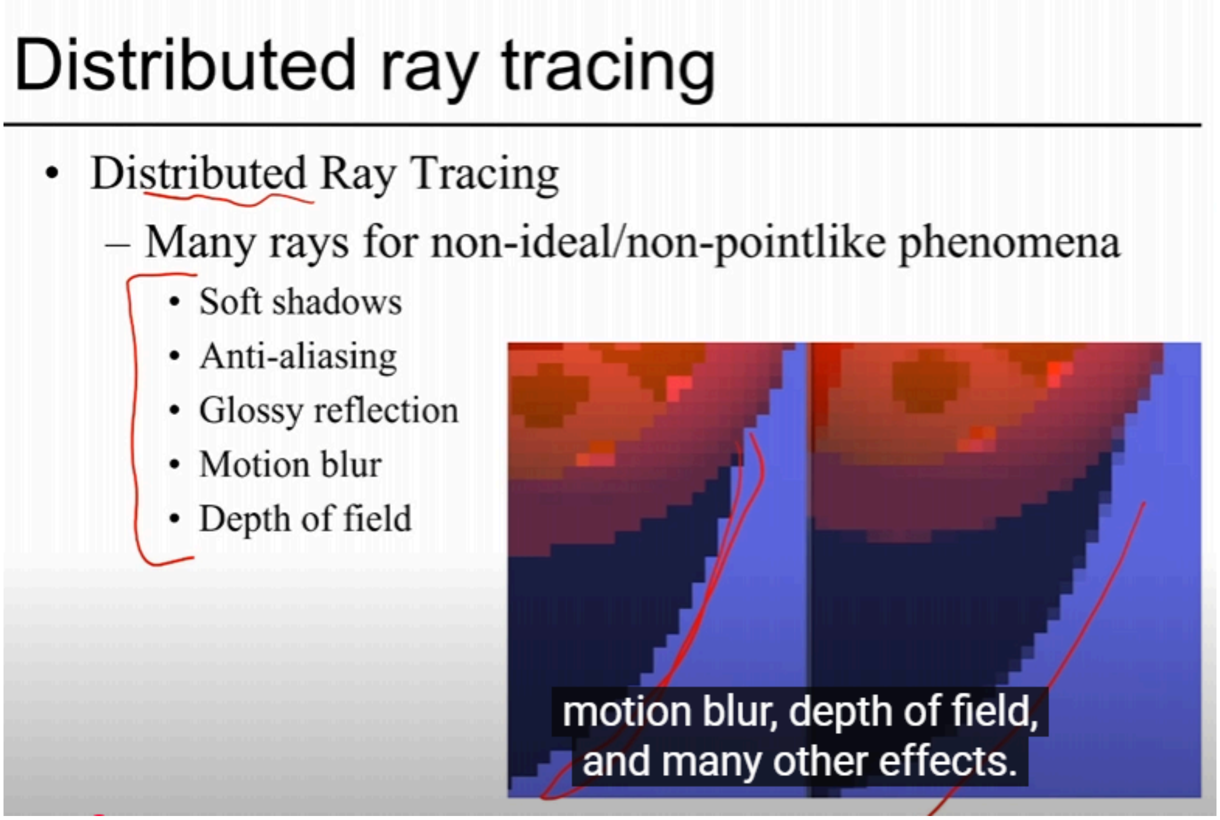
- Hall of mirrors



Recursion For Reflection: 2

that I'm going to leave
you with for next time

- **L12: Accelerating Ray Tracing; bounding volumes, Kd trees**
  - Distributed Ray Tracing

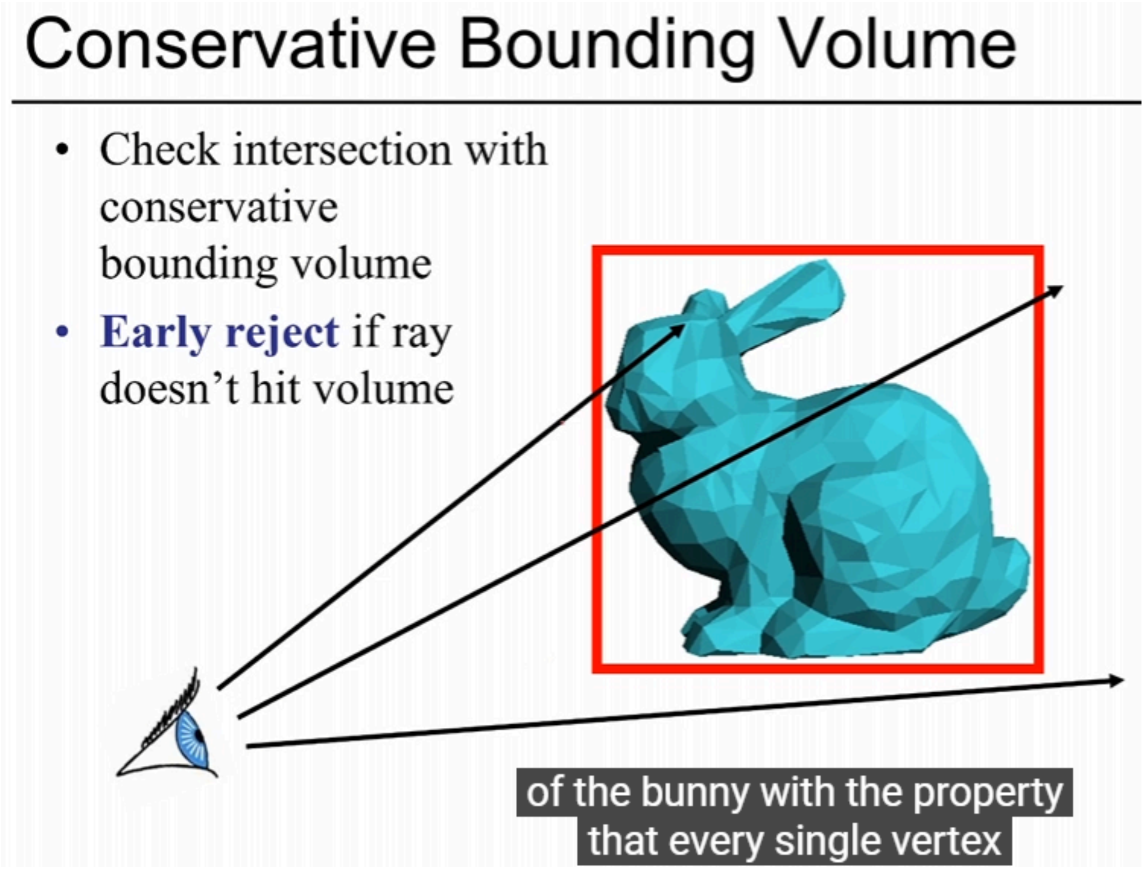


Distributed ray tracing

- Distributed Ray Tracing
  - Many rays for non-ideal/non-pointlike phenomena
    - Soft shadows
    - Anti-aliasing
    - Glossy reflection
    - Motion blur
    - Depth of field

motion blur, depth of field,
and many other effects.

- Bounding Volumes
    - Conservative Bounding Volume

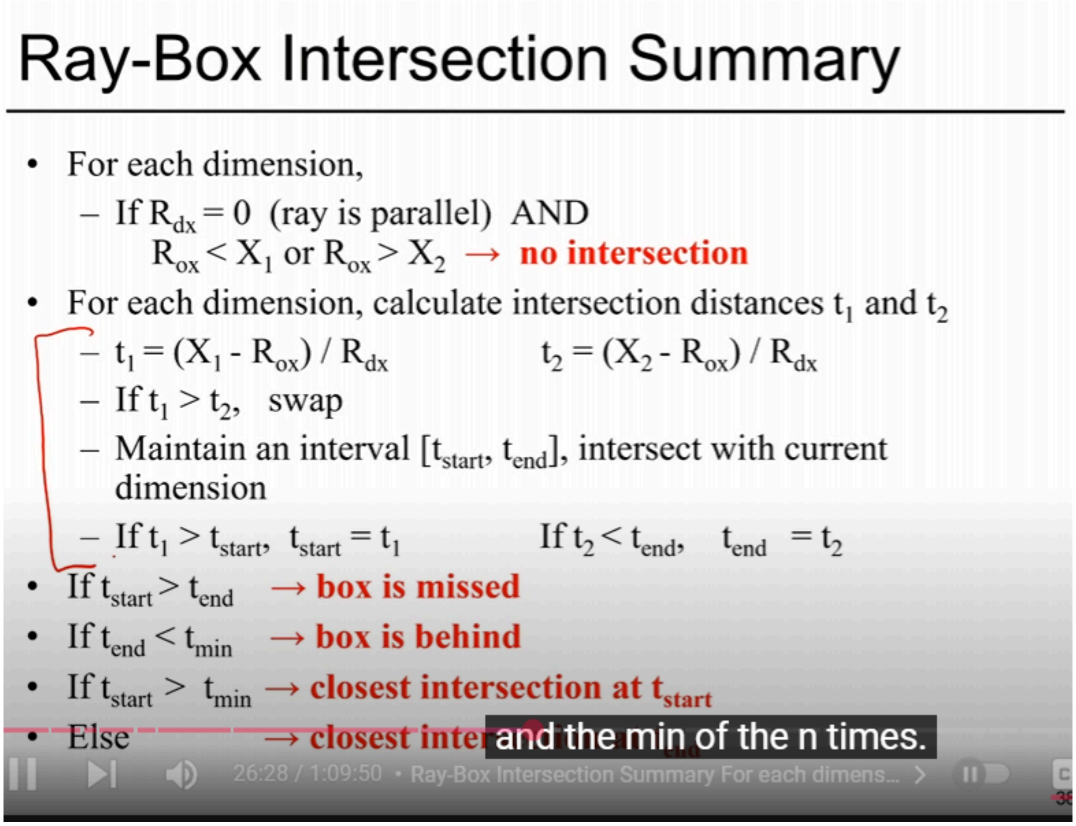


Conservative Bounding Volume

- Check intersection with conservative bounding volume
- **Early reject** if ray doesn't hit volume

of the bunny with the property that every single vertex

- Ray-Box Intersection



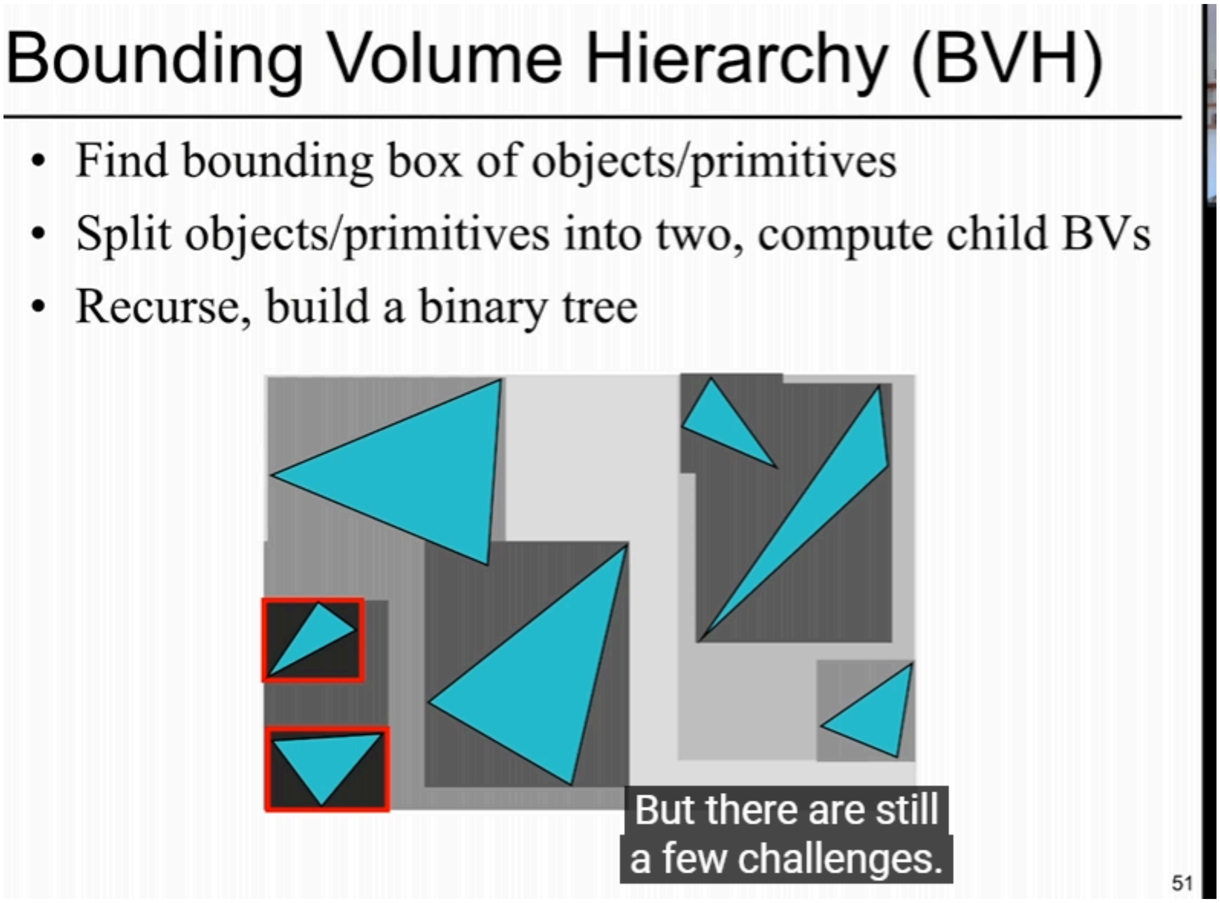
Ray-Box Intersection Summary

- For each dimension,
    - If $R_{dx} = 0$ (ray is parallel) AND $R_{ox} < X_1$ or $R_{ox} > X_2$ → **no intersection**
- For each dimension, calculate intersection distances t_1 and t_2
    - $t_1 = (X_1 - R_{ox}) / R_{dx}$ $t_2 = (X_2 - R_{ox}) / R_{dx}$
    - If $t_1 > t_2$, swap
    - Maintain an interval $[t_{start}, t_{end}]$, intersect with current dimension
    - If $t_1 > t_{start}$, $t_{start} = t_1$ If $t_2 < t_{end}$, $t_{end} = t_2$
- If $t_{start} > t_{end}$ → **box is missed**
- If $t_{end} < t_{min}$ → **box is behind**
- If $t_{start} > t_{min}$ → **closest intersection at t_{start}**
- Else → **closest inter**

and the min of the n times.

26:28 / 1:09:50 • Ray-Box Intersection Summary For each dimens... >

- Bounding Volume Hierarchies (BVH)



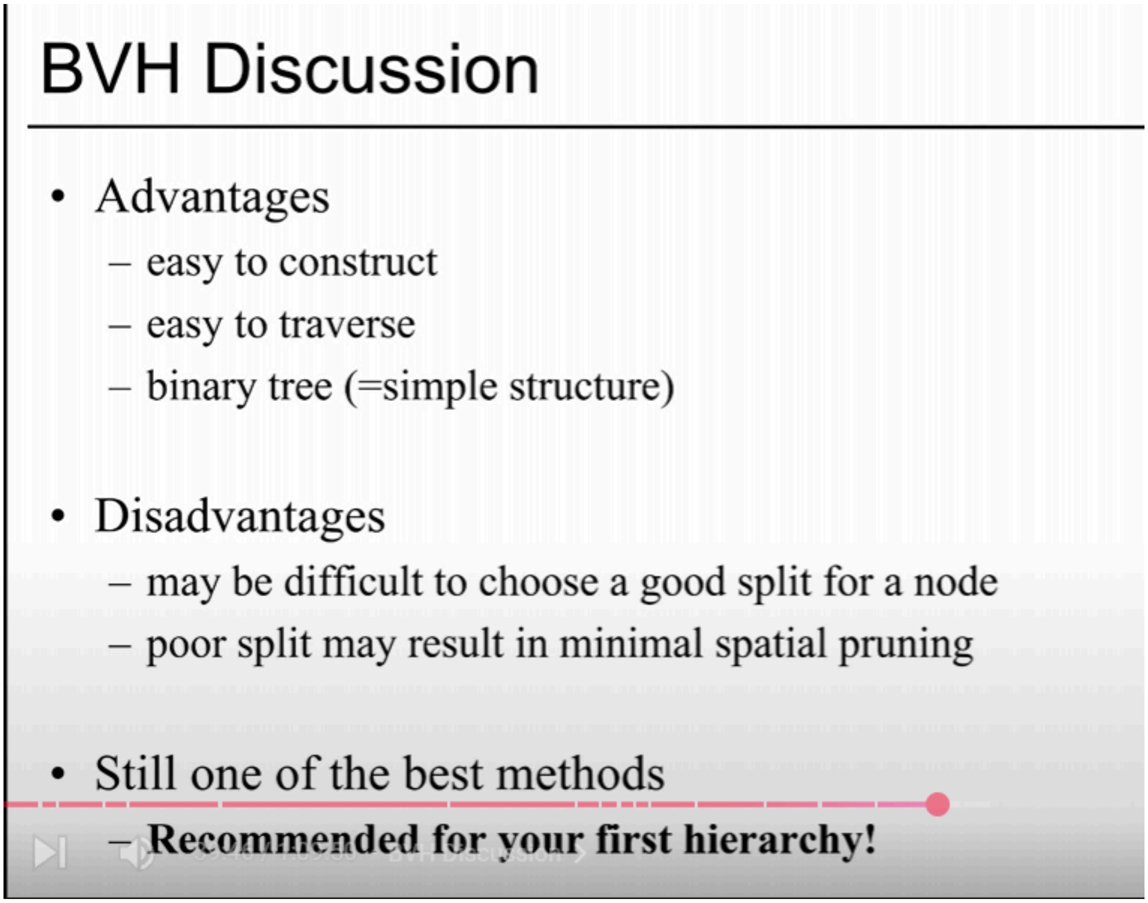
Bounding Volume Hierarchy (BVH)

- Find bounding box of objects/primitives
- Split objects/primitives into two, compute child BVs
- Recurse, build a binary tree

But there are still a few challenges.

51

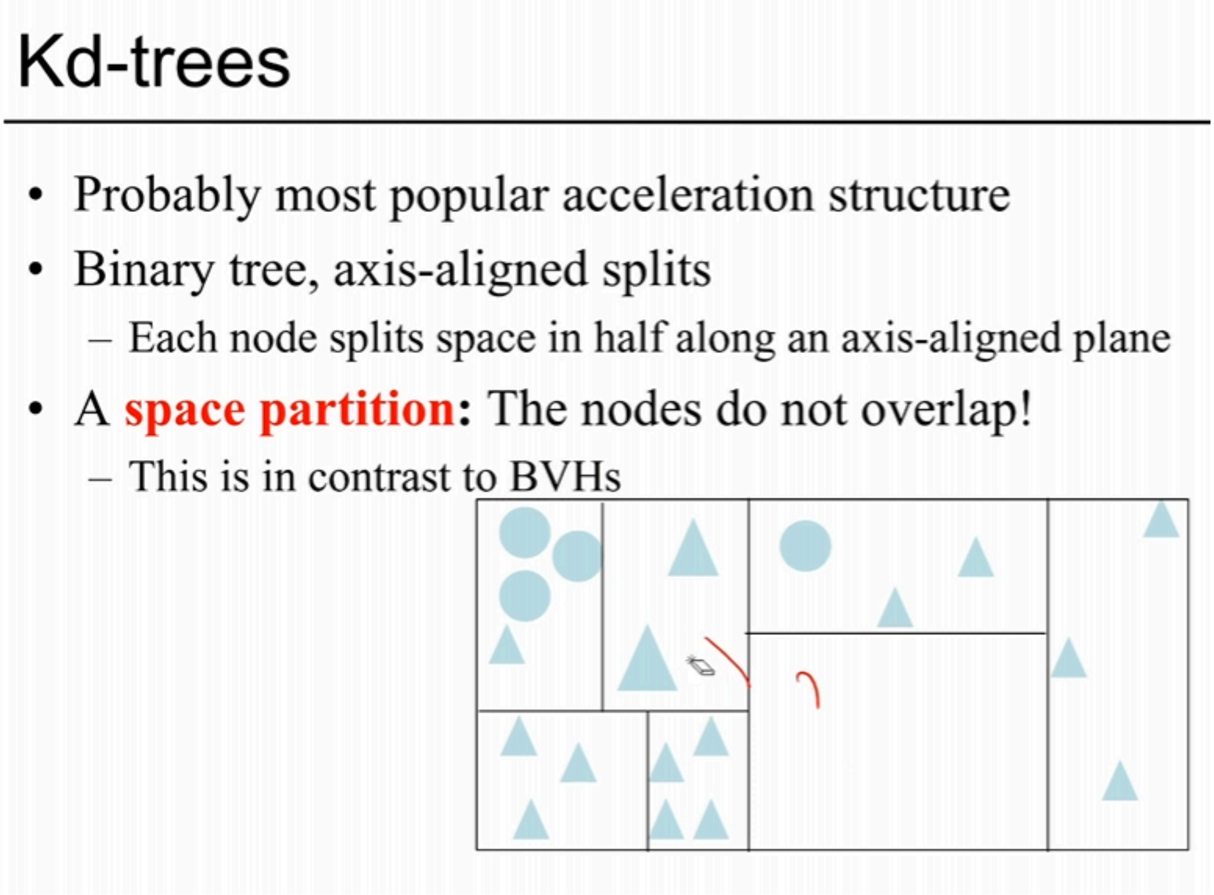
  - Pros and Cons



BVH Discussion

- Advantages
  - easy to construct
  - easy to traverse
  - binary tree (=simple structure)
- Disadvantages
  - may be difficult to choose a good split for a node
  - poor split may result in minimal spatial pruning
- Still one of the best methods
  - **Recommended for your first hierarchy!**

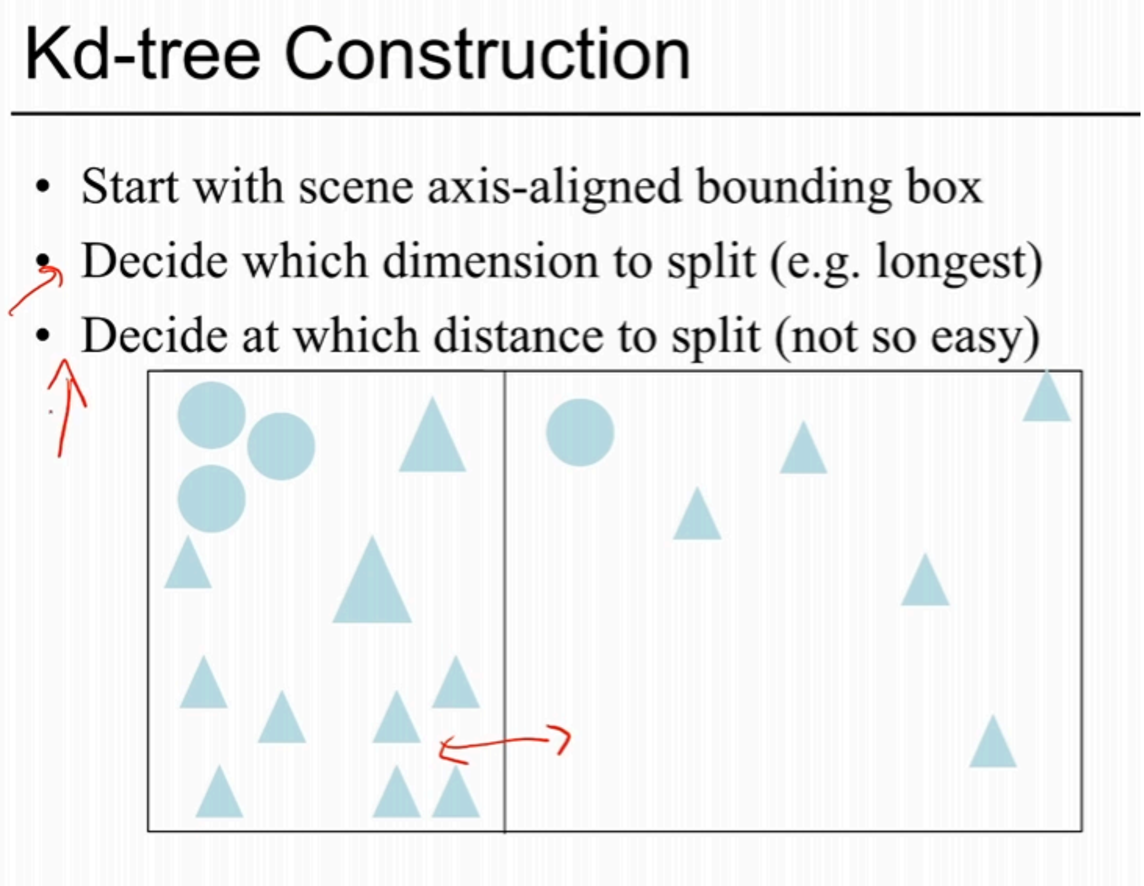
- Kd-trees



Kd-trees

- Probably most popular acceleration structure
- Binary tree, axis-aligned splits
  - Each node splits space in half along an axis-aligned plane
- A **space partition:** The nodes do not overlap!
  - This is in contrast to BVHs

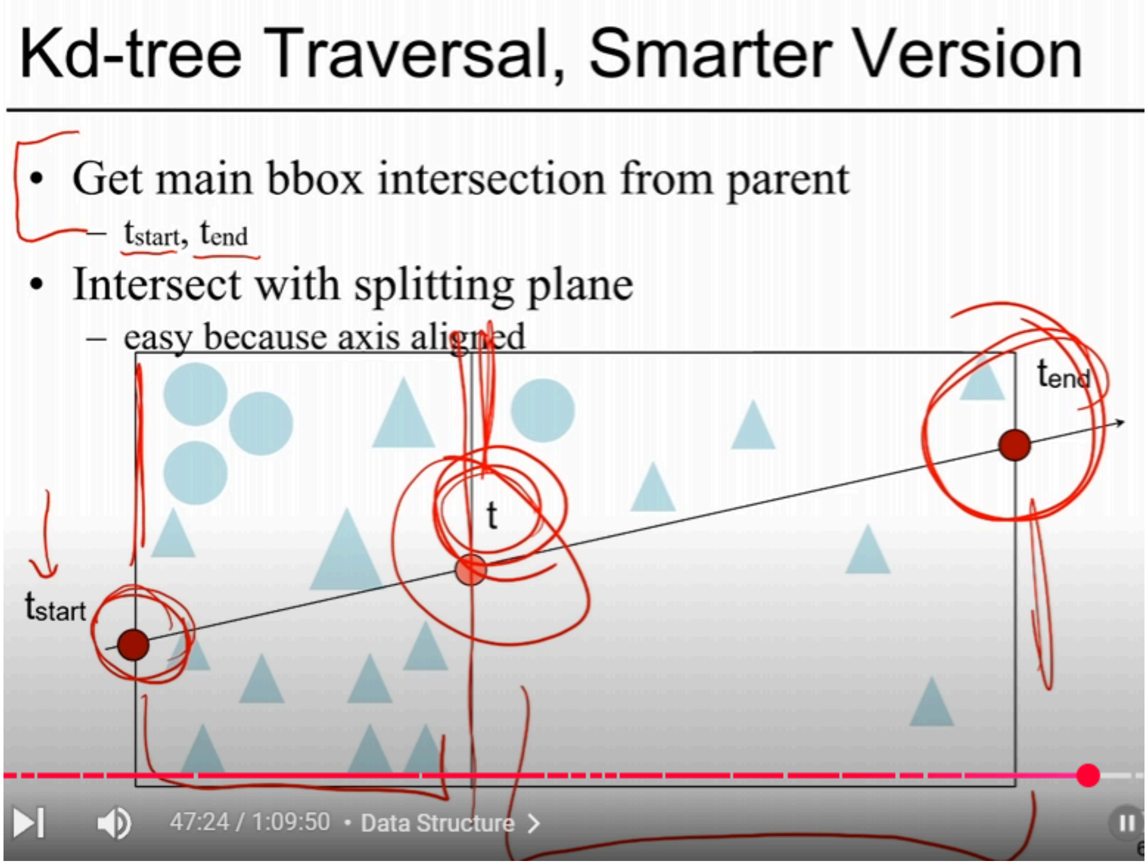
  - Construction



Kd-tree Construction

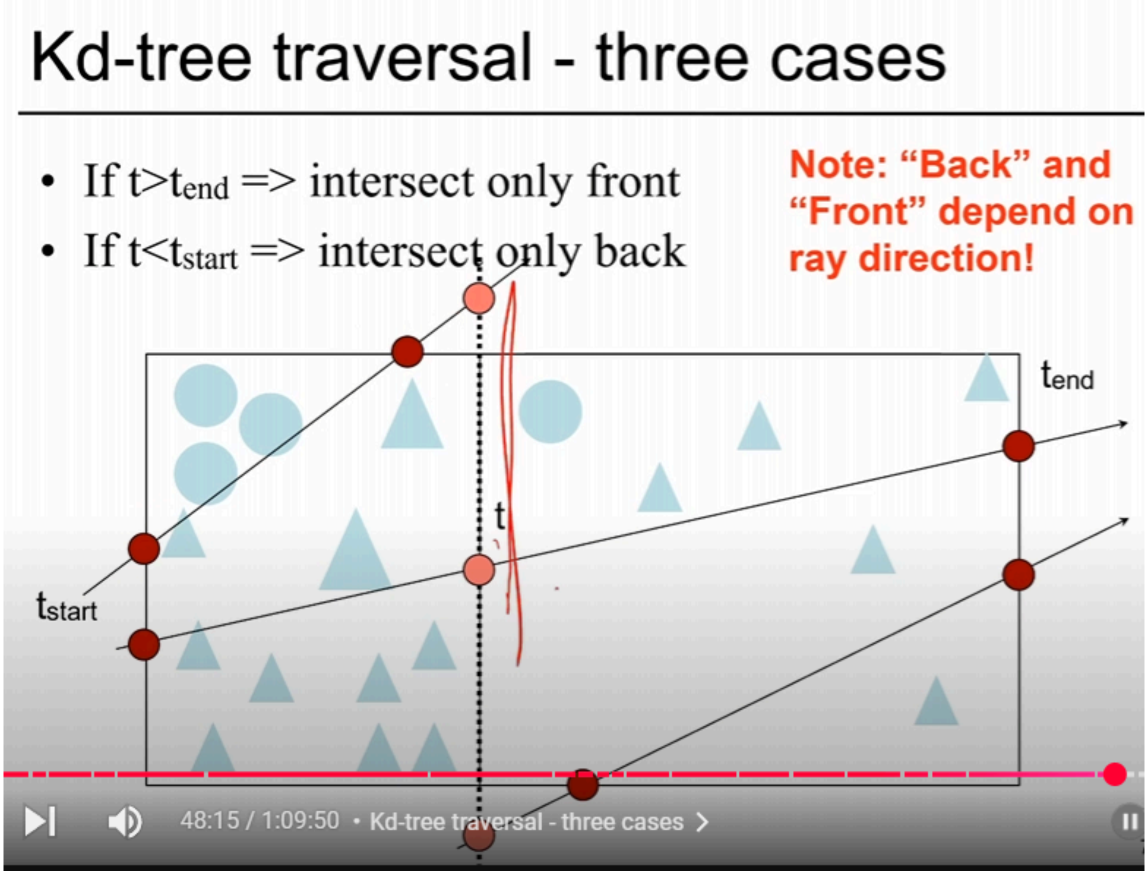
- Start with scene axis-aligned bounding box
- Decide which dimension to split (e.g. longest)
- Decide at which distance to split (not so easy)

- Traversal



Kd-tree Traversal, Smarter Version

- Get main bbox intersection from parent
  - t_{start}, t_{end}
- Intersect with splitting plane
  - easy because axis aligned

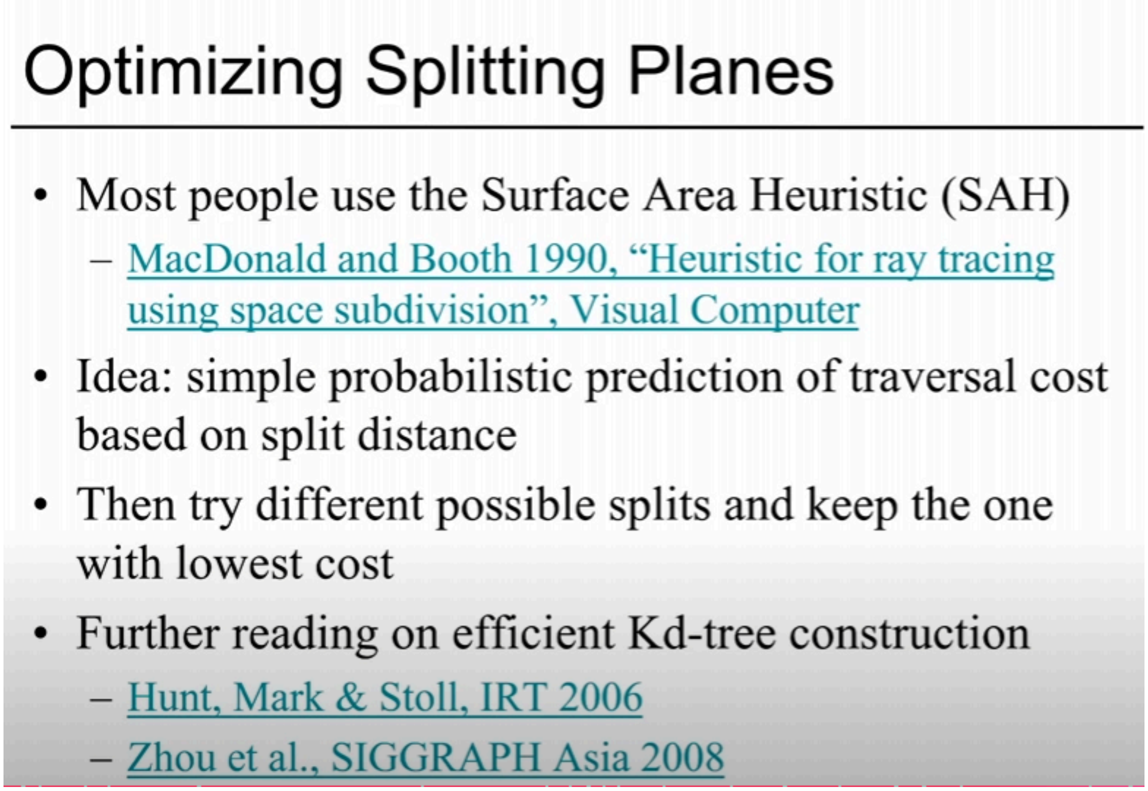


Kd-tree traversal - three cases

- If $t > t_{end}$ => intersect only front
- If $t < t_{start}$ => intersect only back

Note: "Back" and "Front" depend on ray direction!

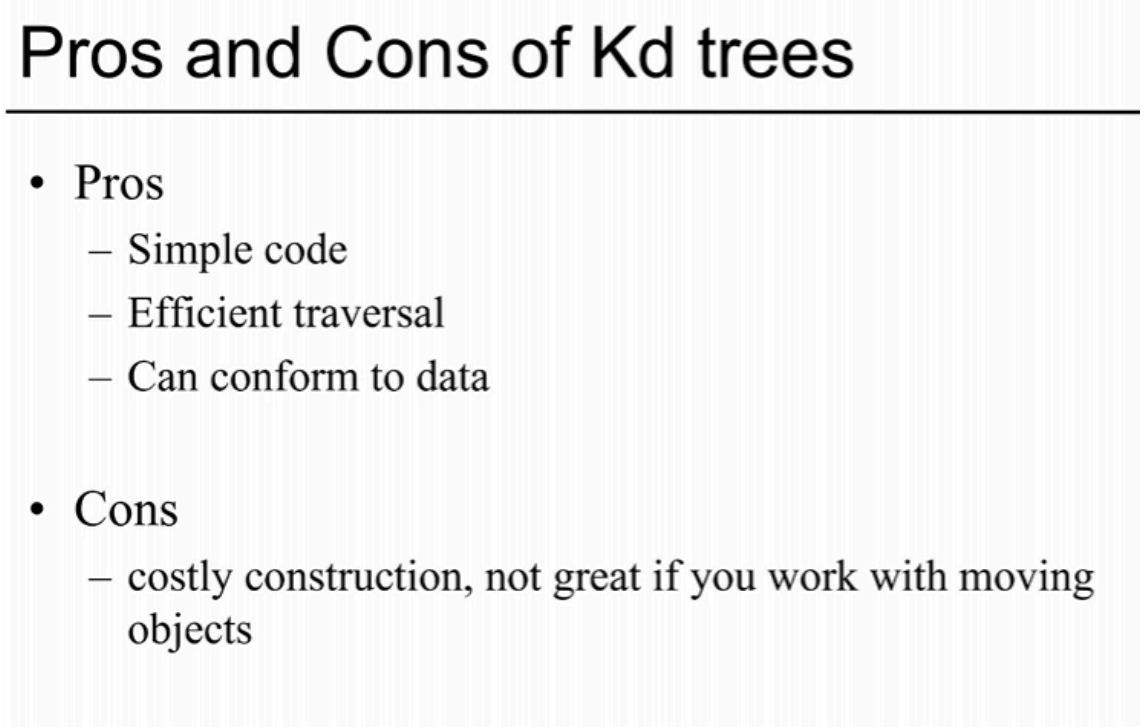
- Optimizing Splitting Planes



Optimizing Splitting Planes

- Most people use the Surface Area Heuristic (SAH)
  - MacDonald and Booth 1990, “Heuristic for ray tracing using space subdivision”, Visual Computer
- Idea: simple probabilistic prediction of traversal cost based on split distance
- Then try different possible splits and keep the one with lowest cost
- Further reading on efficient Kd-tree construction
  - Hunt, Mark & Stoll, IRT 2006
  - Zhou et al., SIGGRAPH Asia 2008

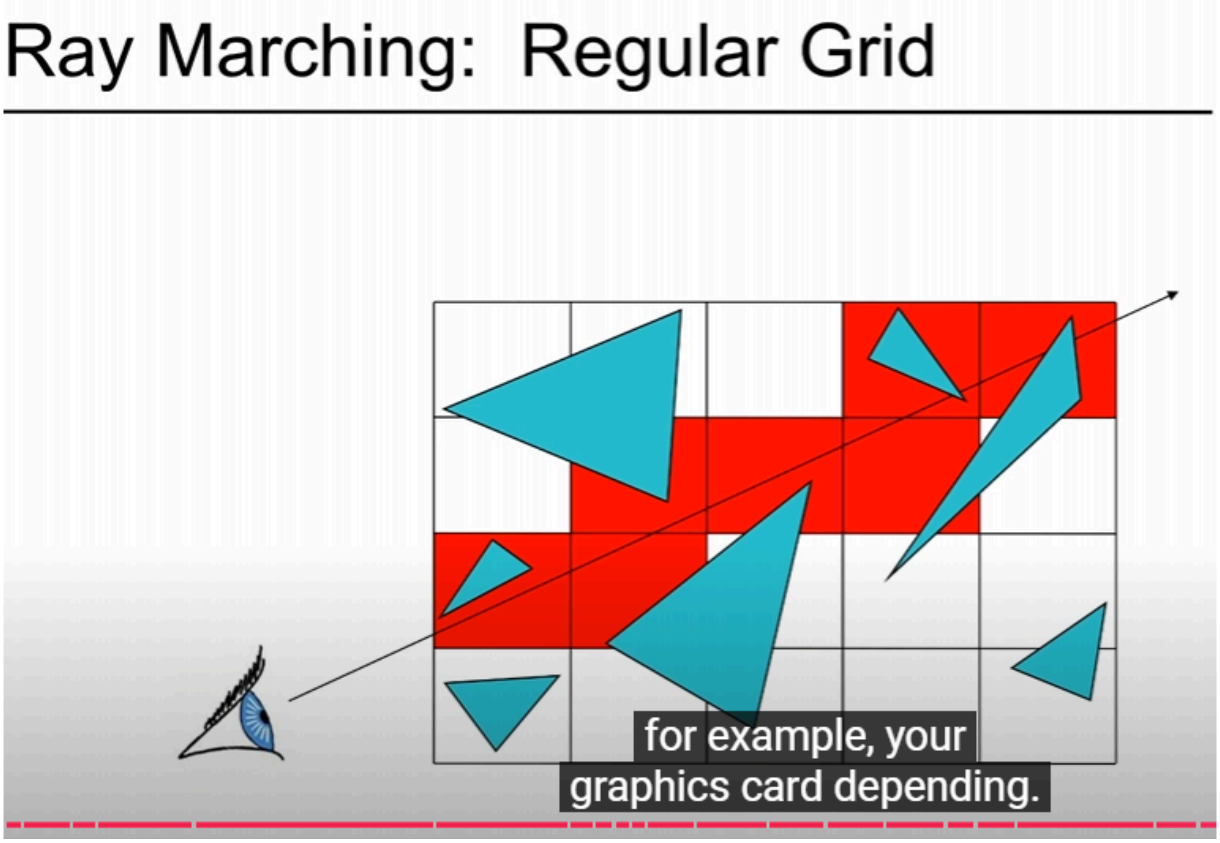
- Pros and Cons



Pros and Cons of Kd trees

- Pros
  - Simple code
  - Efficient traversal
  - Can conform to data
- Cons
  - costly construction, not great if you work with moving objects

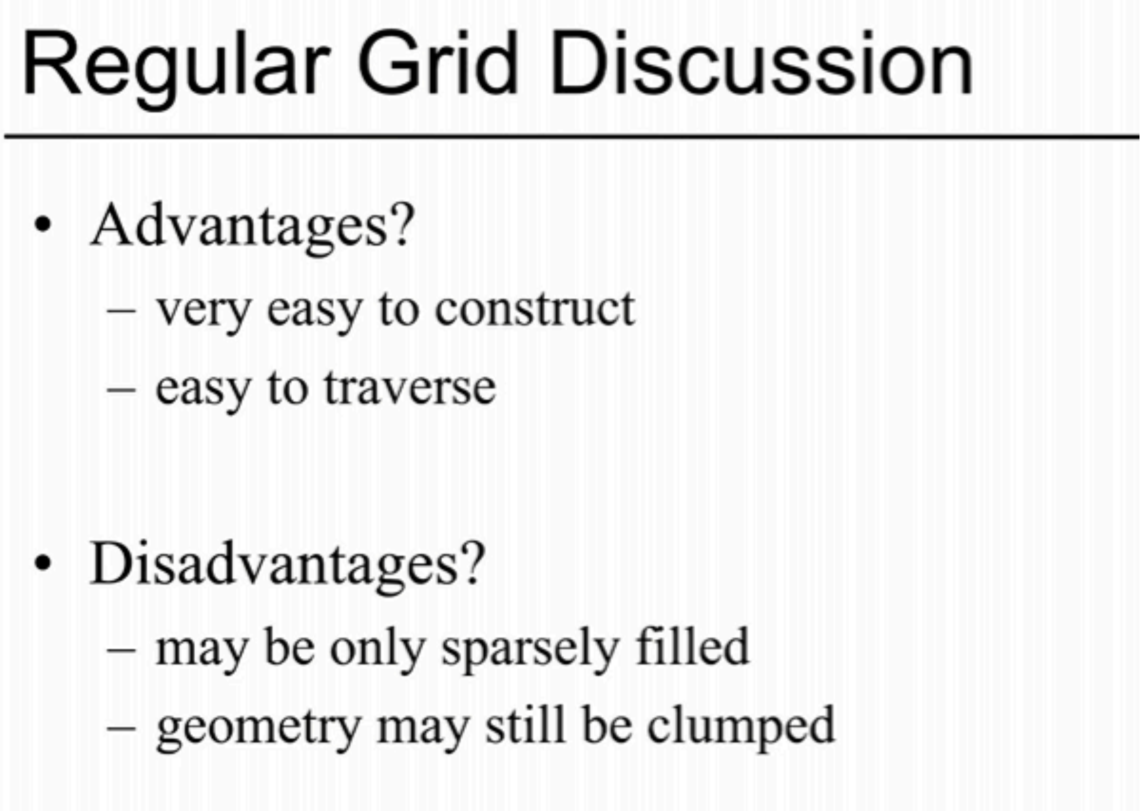
- Ray Marching: Regular Grid



Ray Marching: Regular Grid

for example, your
graphics card depending.

  - Pros and Cons

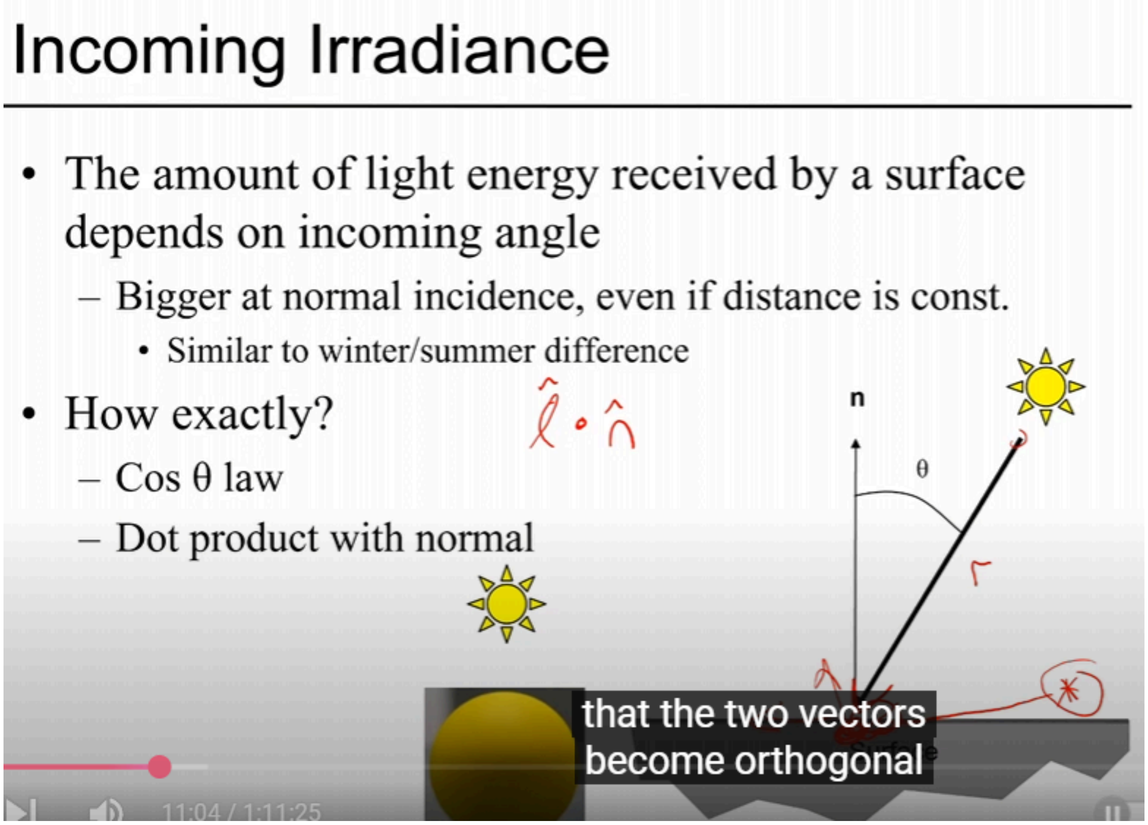


Regular Grid Discussion

- Advantages?
  - very easy to construct
  - easy to traverse
- Disadvantages?
  - may be only sparsely filled
  - geometry may still be clumped

- **L13: Shading and Materials**
  - Lighting and Material Apperance
    - Input for realistic rendering

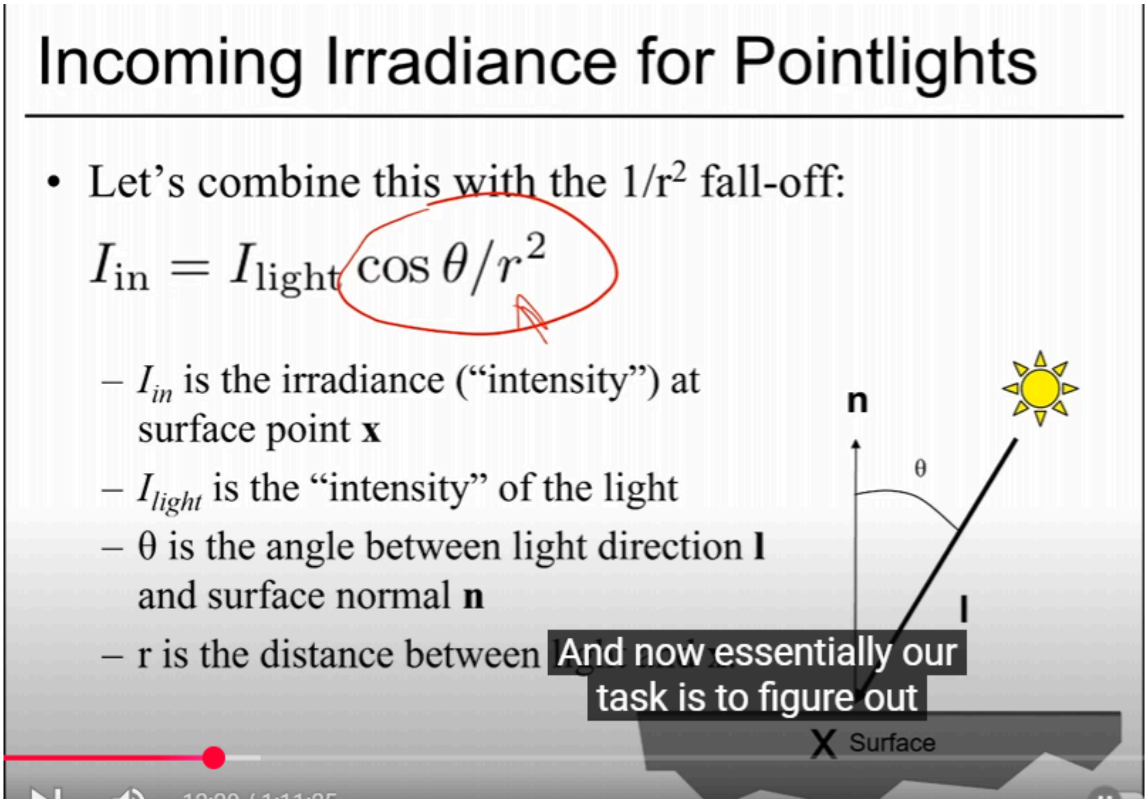
        - Geometry, lighting and materials
    - Material apperance
        - Intensity and shape of highlights
        - Glossiness
        - Color
        - Spatial variation, i.e., Texture
- Light Sources
    - Incoming Irradiance



Incoming Irradiance

- The amount of light energy received by a surface depends on incoming angle
  - Bigger at normal incidence, even if distance is const.
    - Similar to winter/summer difference
- How exactly? $\hat{l} \cdot \hat{n}$
  - Cos θ law
  - Dot product with normal

that the two vectors
become orthogonal



Incoming Irradiance for Pointlights

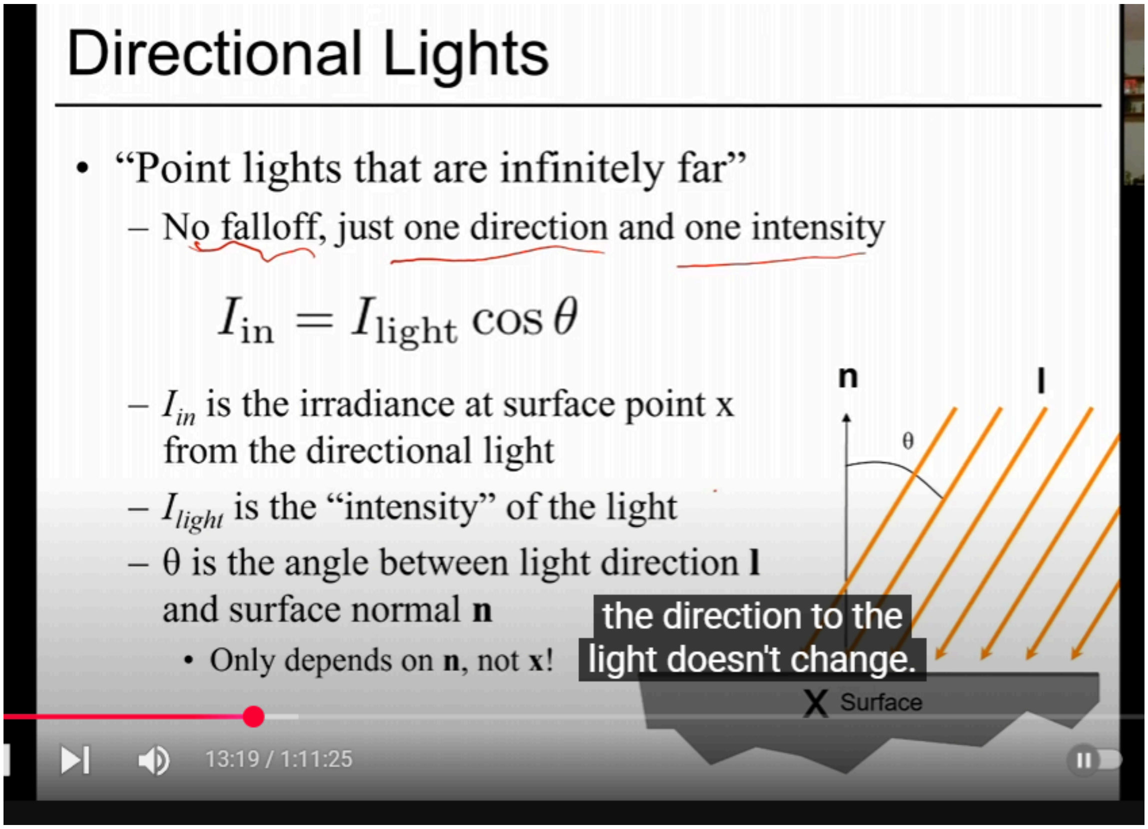
- Let's combine this with the 1/r² fall-off:

$$I_{\text{in}} = I_{\text{light}} \boxed{\cos\theta / r^2}$$

  - I_{in} is the irradiance ("intensity") at surface point **x**
  - I_{light} is the "intensity" of the light
  - θ is the angle between light direction **l** and surface normal **n**
  - r is the distance between

And now essentially our task is to figure out

- Directional Lights



Directional Lights

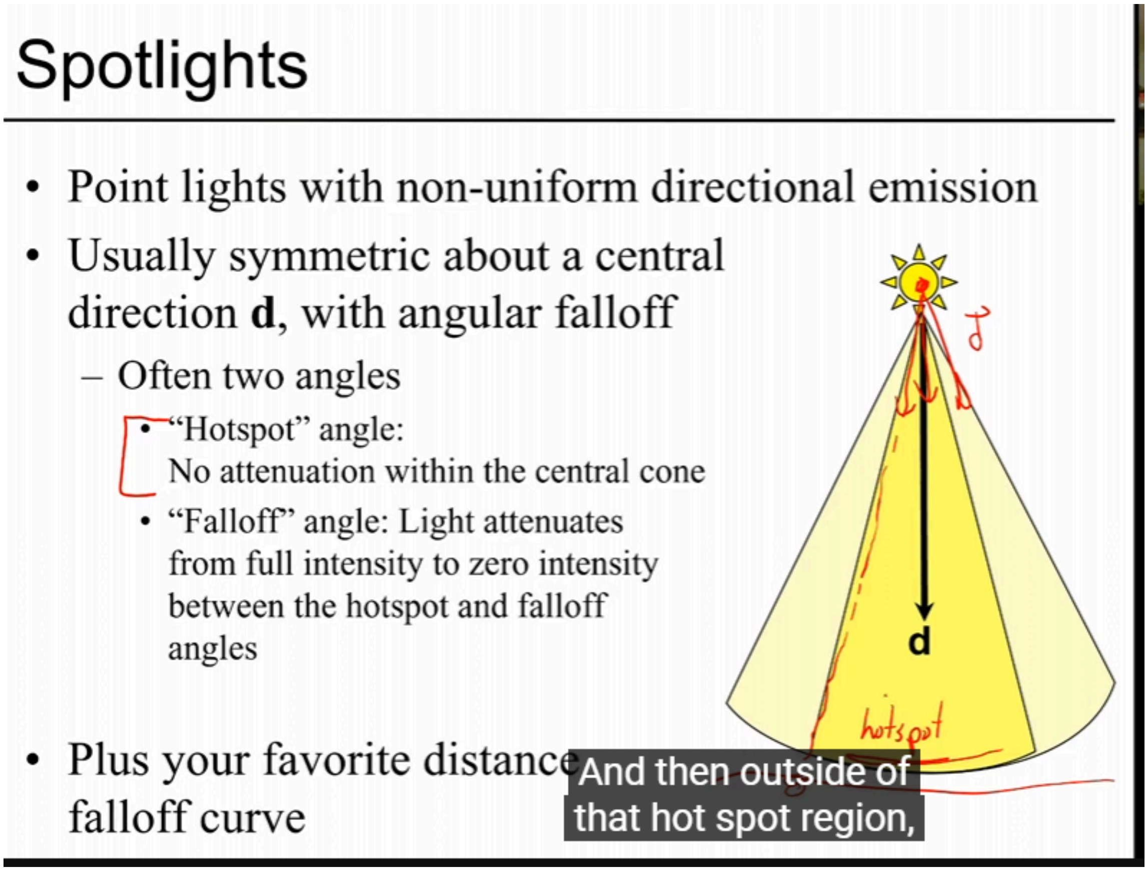
- "Point lights that are infinitely far"
  - No falloff, just one direction and one intensity

$$I_{\text{in}} = I_{\text{light}} \cos\theta$$

  - I_{in} is the irradiance at surface point x from the directional light
  - I_{light} is the "intensity" of the light
  - θ is the angle between light direction **l** and surface normal **n**
    - Only depends on **n**, not **x**!

the direction to the light doesn't change.

- Spotlights

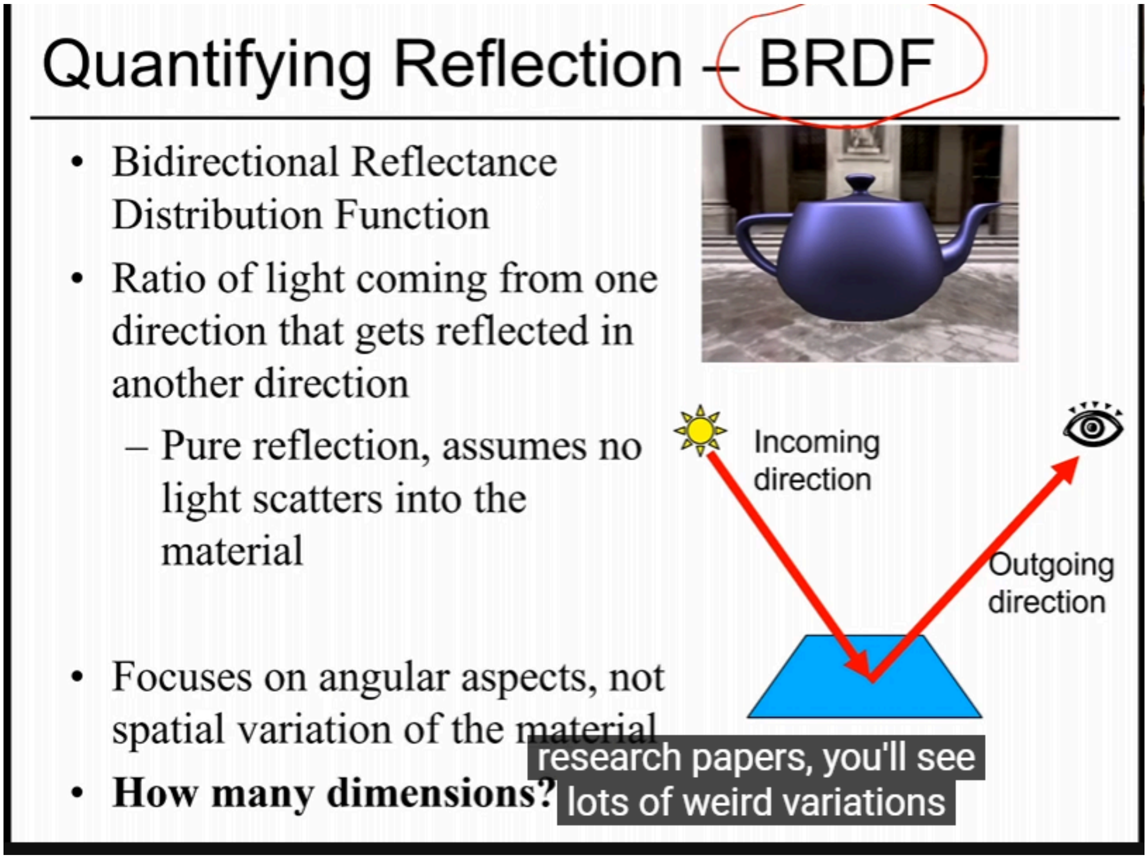


Spotlights

- Point lights with non-uniform directional emission
- Usually symmetric about a central direction **d**, with angular falloff
  - Often two angles
    - "Hotspot" angle: No attenuation within the central cone
    - "Falloff" angle: Light attenuates from full intensity to zero intensity between the hotspot and falloff angles
- Plus your favorite distance falloff curve

And then outside of that hot spot region,

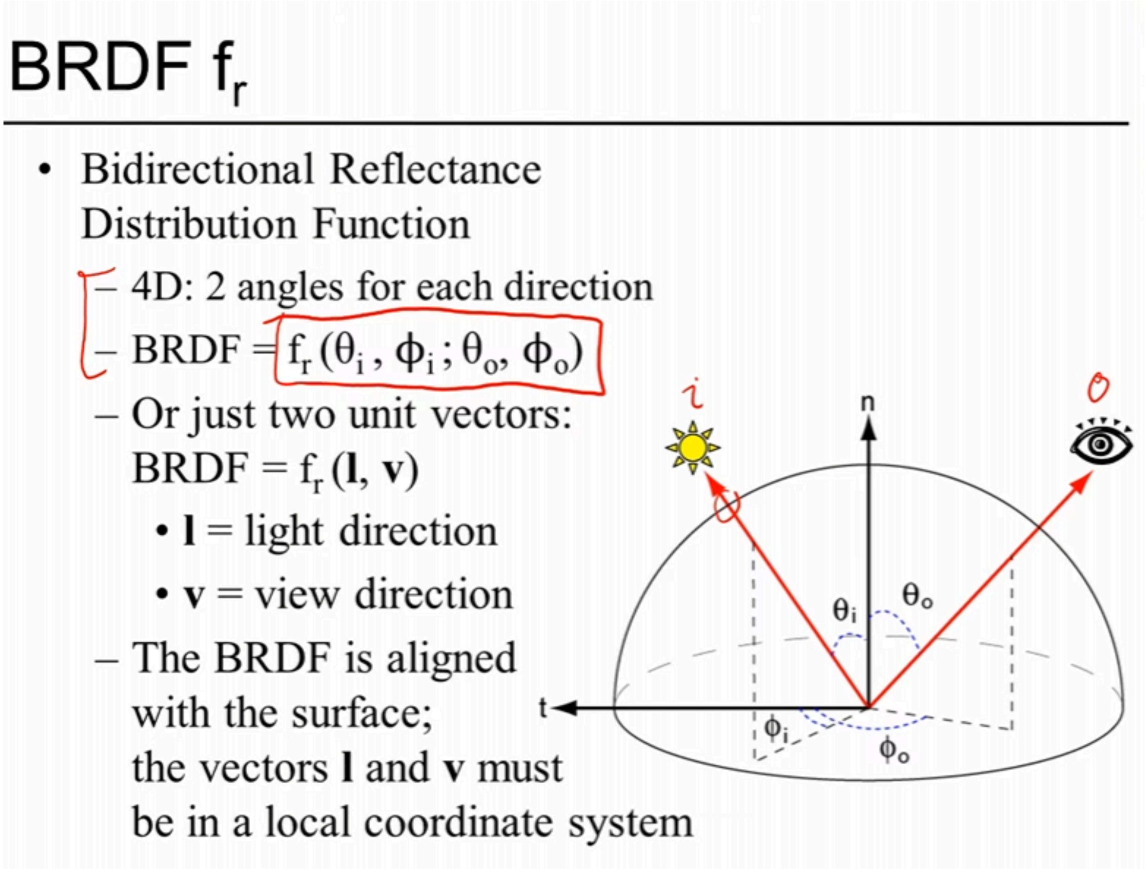
- Quantifying Reflection - BRDF



Quantifying Reflection – BRDF

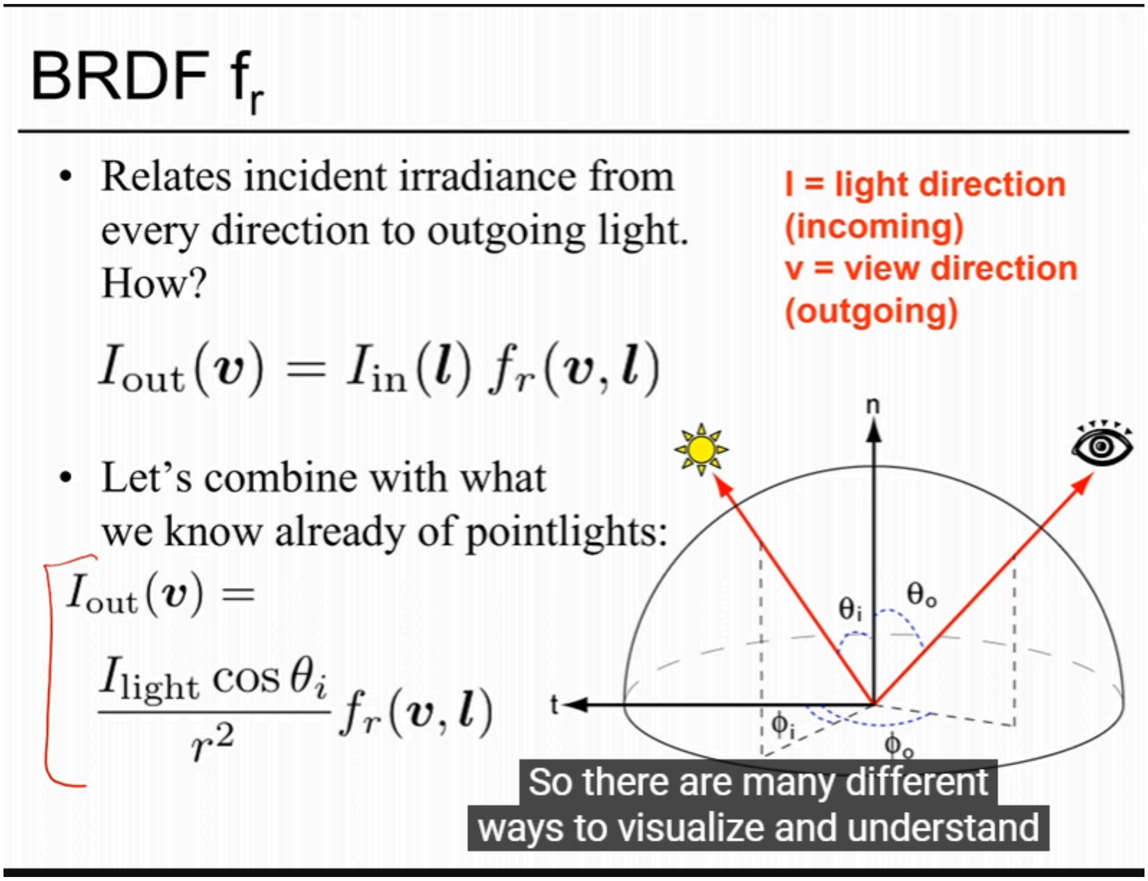
- Bidirectional Reflectance Distribution Function
- Ratio of light coming from one direction that gets reflected in another direction
  - Pure reflection, assumes no light scatters into the material
- Focuses on angular aspects, not spatial variation of the material
- **How many dimensions?**

research papers, you'll see lots of weird variations



BRDF f_r

- Bidirectional Reflectance Distribution Function
  - 4D: 2 angles for each direction
  - BRDF = $\boxed{f_r(\theta_i, \phi_i; \theta_o, \phi_o)}$
  - Or just two unit vectors: BRDF = $f_r(\mathbf{l}, \mathbf{v})$
    - **l** = light direction
    - **v** = view direction
  - The BRDF is aligned with the surface; the vectors **l** and **v** must be in a local coordinate system



BRDF f_r

- Relates incident irradiance from every direction to outgoing light. How?

$$I_{\text{out}}(\boldsymbol{v}) = I_{\text{in}}(\boldsymbol{l})\, f_r(\boldsymbol{v}, \boldsymbol{l})$$

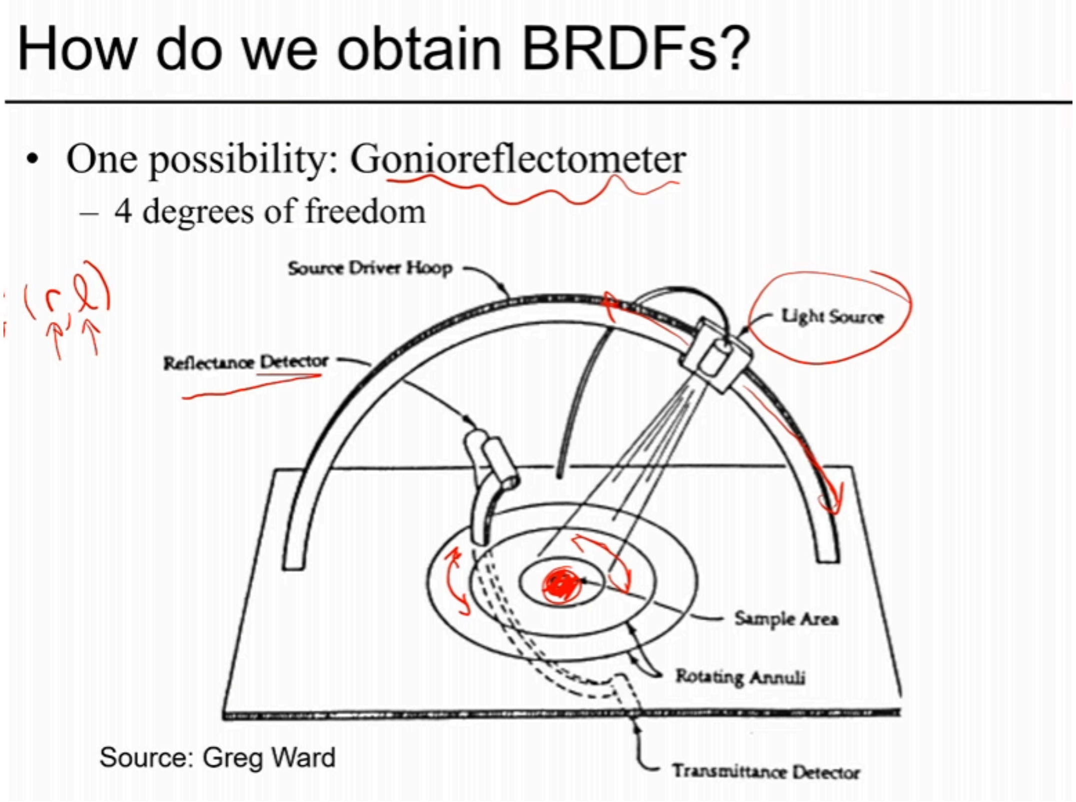
- Let's combine with what we know already of pointlights:

$$I_{\text{out}}(\boldsymbol{v}) = \frac{I_{\text{light}} \cos\theta_i}{r^2} f_r(\boldsymbol{v}, \boldsymbol{l})$$

l = light direction (incoming)
v = view direction (outgoing)

So there are many different ways to visualize and understand

- Obtain BRDF

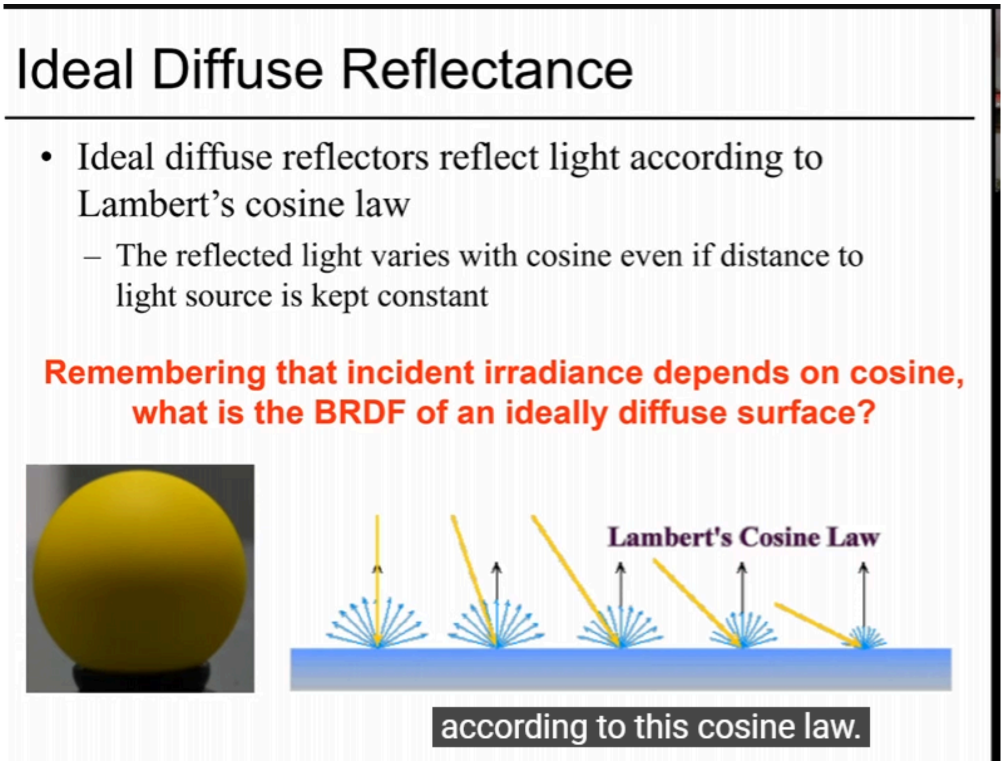


How do we obtain BRDFs?

- One possibility: Gonioreflectometer
  - 4 degrees of freedom

Source: Greg Ward

- Parametric BRDFs
  - Ideal Diffuse Reflectance

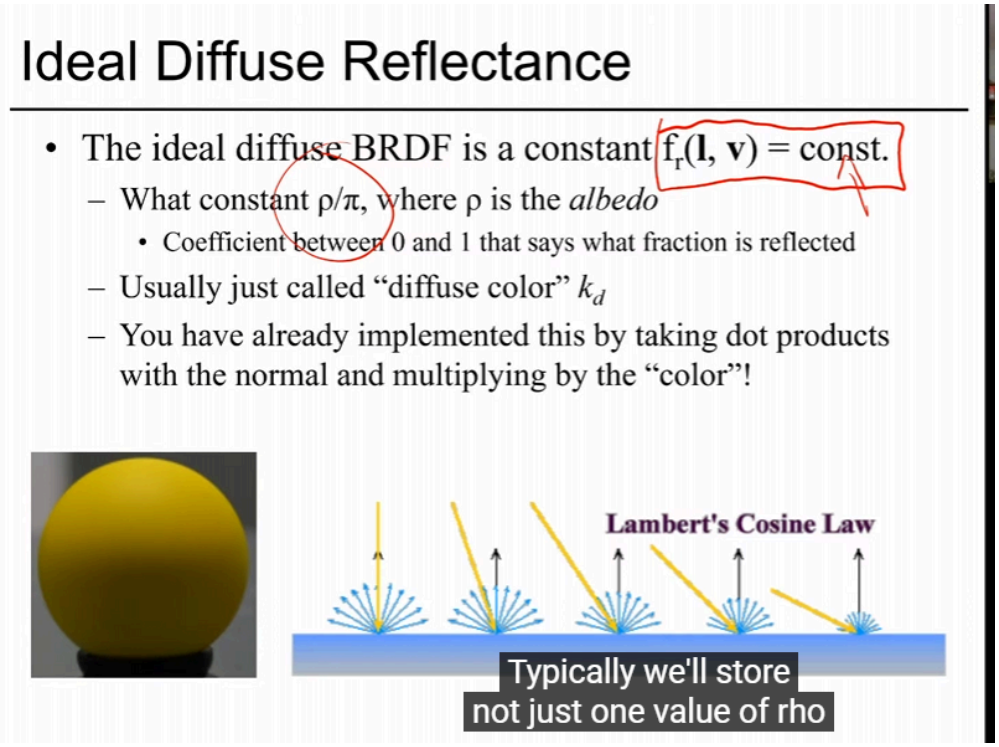


Ideal Diffuse Reflectance

- Ideal diffuse reflectors reflect light according to Lambert's cosine law
  - The reflected light varies with cosine even if distance to light source is kept constant

Remembering that incident irradiance depends on cosine, what is the BRDF of an ideally diffuse surface?

according to this cosine law.

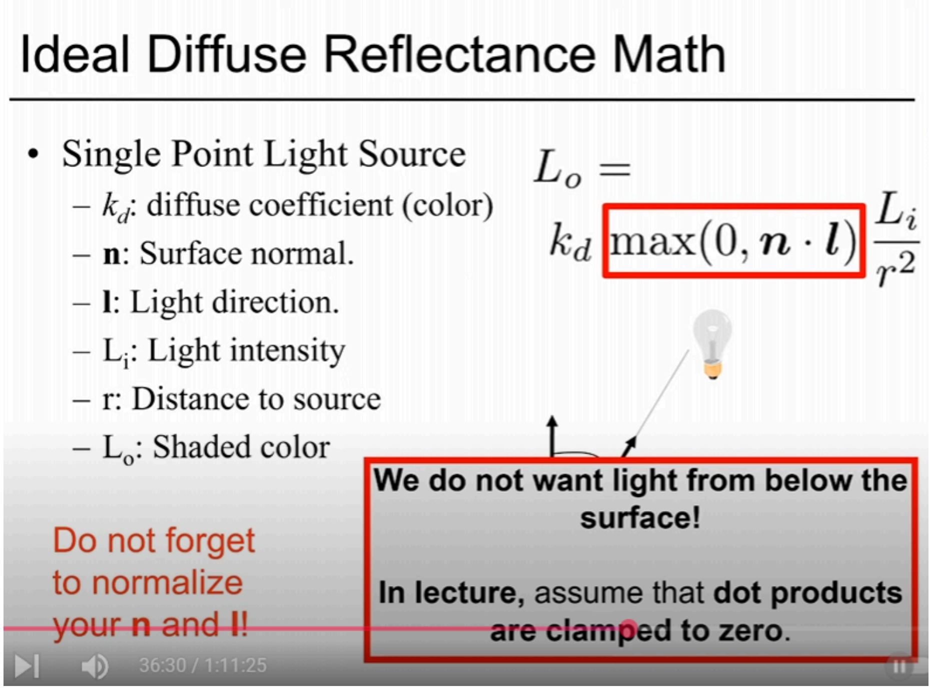


Ideal Diffuse Reflectance

- The ideal diffuse BRDF is a constant $f_r(\mathbf{l}, \mathbf{v}) = \text{const.}$
  - What constant ρ/π, where ρ is the *albedo*
    - Coefficient between 0 and 1 that says what fraction is reflected
  - Usually just called "diffuse color" k_d
  - You have already implemented this by taking dot products with the normal and multiplying by the "color"!

Typically we'll store not just one value of rho

- Albedo = 0, Absorb all light, Albedo increse, more light relfected
- Math



Ideal Diffuse Reflectance Math

- Single Point Light Source
  - k_d: diffuse coefficient (color)
  - $\mathbf{n}$: Surface normal.
  - $\mathbf{l}$: Light direction.
  - L_i: Light intensity
  - r: Distance to source
  - L_o: Shaded color

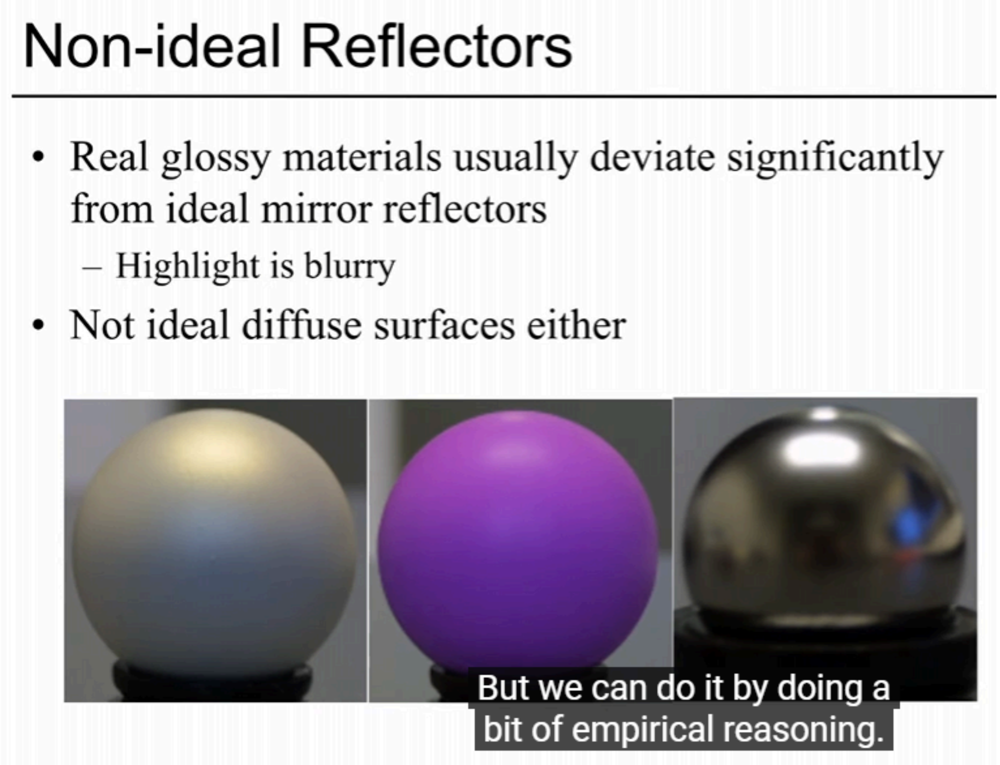
$$L_o = k_d \boxed{\max(0, \boldsymbol{n} \cdot \boldsymbol{l})} \frac{L_i}{r^2}$$

Do not forget to normalize your **n** and **l**!

We do not want light from below the surface!

In lecture, assume that **dot products are clamped to zero.**

- Non-ideal Reflectors

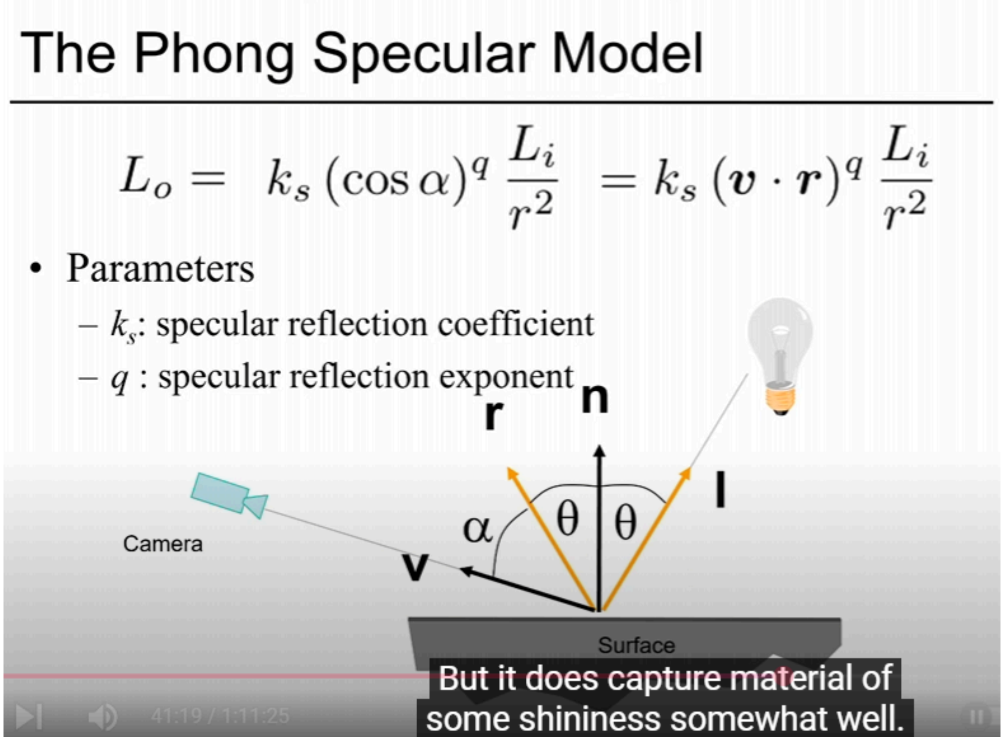


Non-ideal Reflectors

- Real glossy materials usually deviate significantly from ideal mirror reflectors
  - Highlight is blurry
- Not ideal diffuse surfaces either

But we can do it by doing a bit of empirical reasoning.

- The Phong Specular Model



The Phong Specular Model

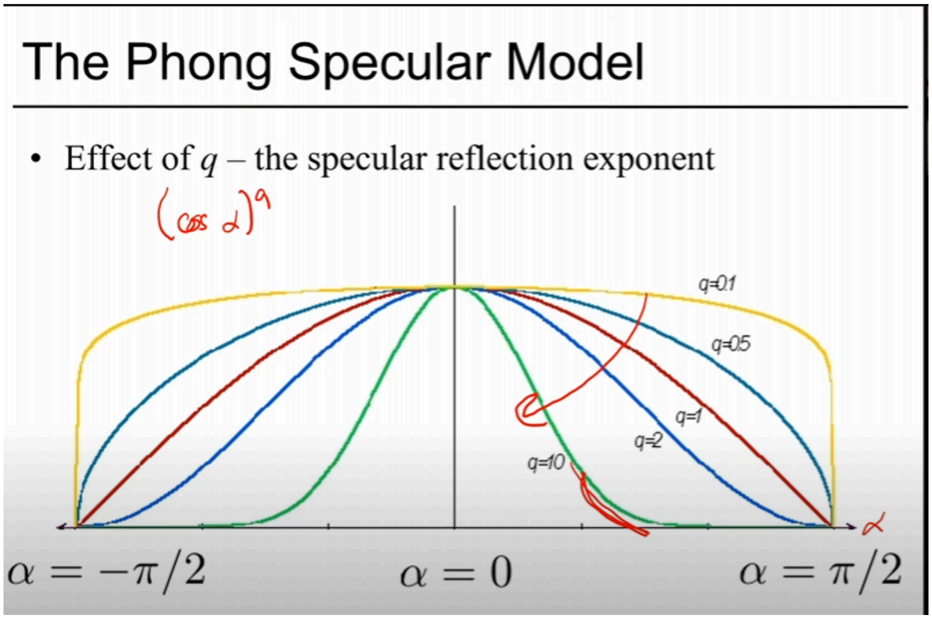
$$L_o = k_s \left(\cos\alpha\right)^q \frac{L_i}{r^2} = k_s \left(\boldsymbol{v}\cdot\boldsymbol{r}\right)^q \frac{L_i}{r^2}$$

- Parameters
  - k_s: specular reflection coefficient
  - q : specular reflection exponent

But it does capture material of some shininess somewhat well.

  - if a = 0, then reflect all light

- q: how sharply it drop off

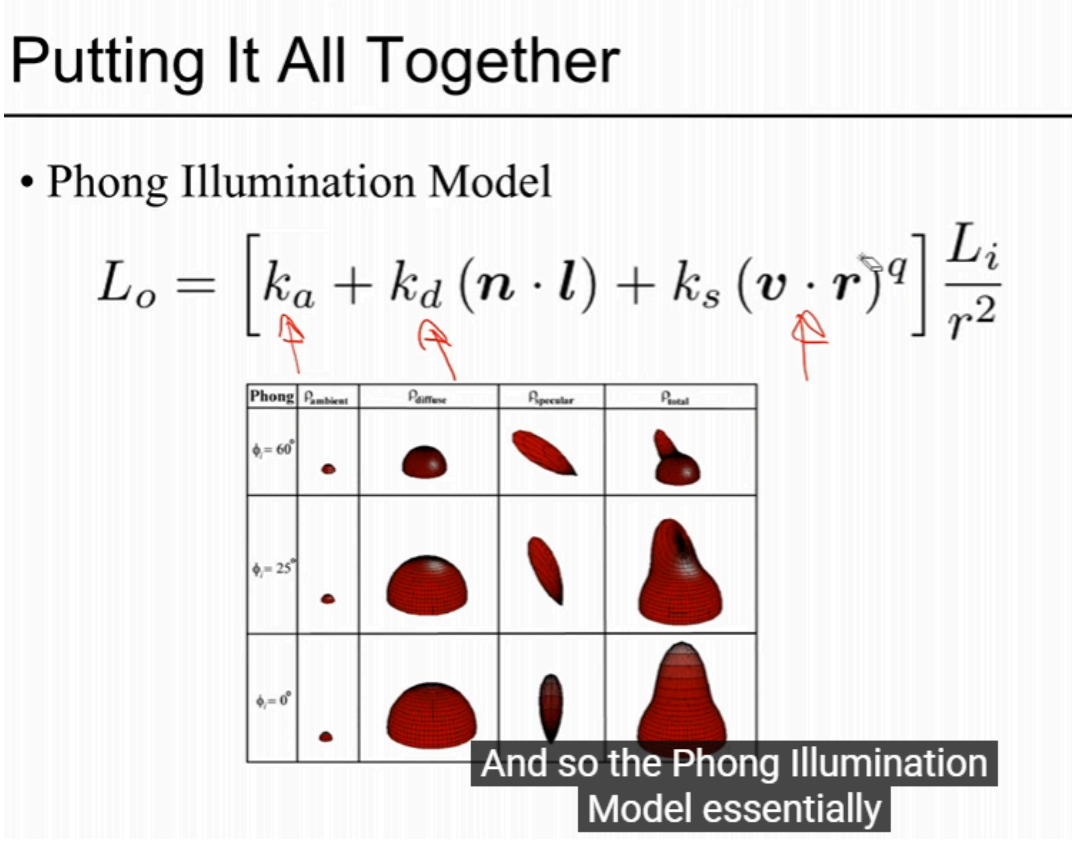


The Phong Specular Model

- Effect of q – the specular reflection exponent

$(\cos \alpha)^q$

- Ambient Illumination
  - Phong Illumination Model



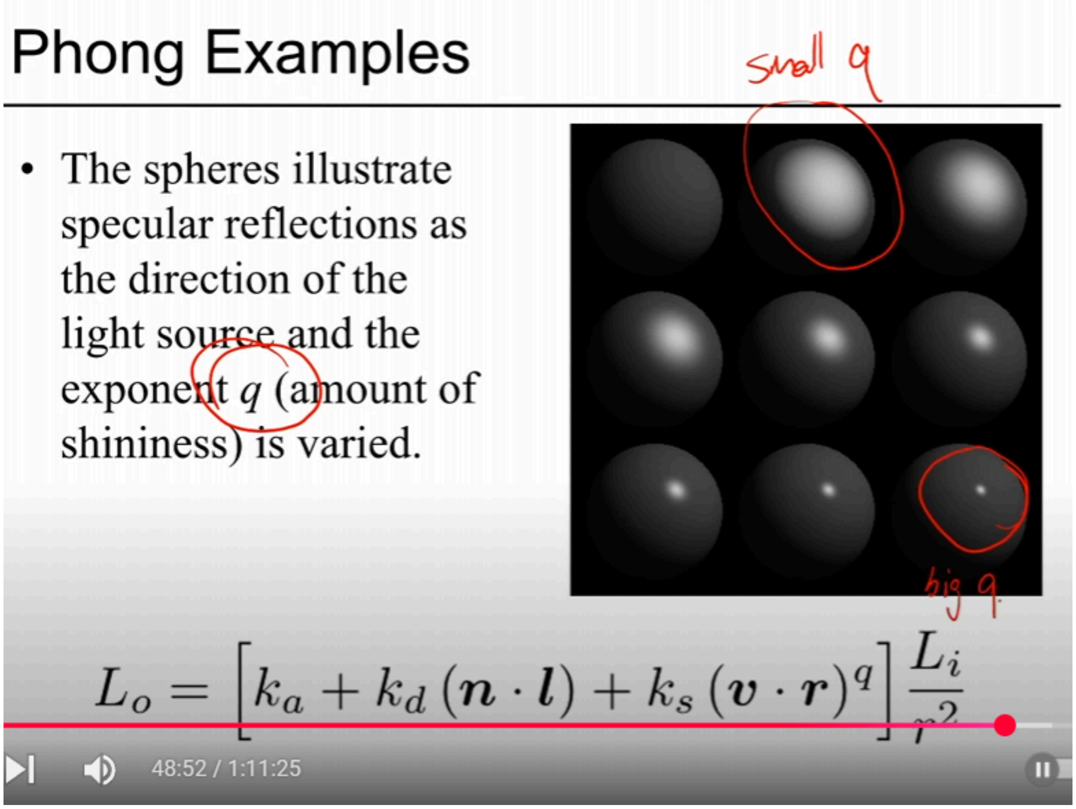
Putting It All Together

- Phong Illumination Model

$$L_o = \left[k_a + k_d\,(\boldsymbol{n} \cdot \boldsymbol{l}) + k_s\,(\boldsymbol{v} \cdot \boldsymbol{r})^q\right] \frac{L_i}{r^2}$$

And so the Phong Illumination Model essentially

- Phong Example

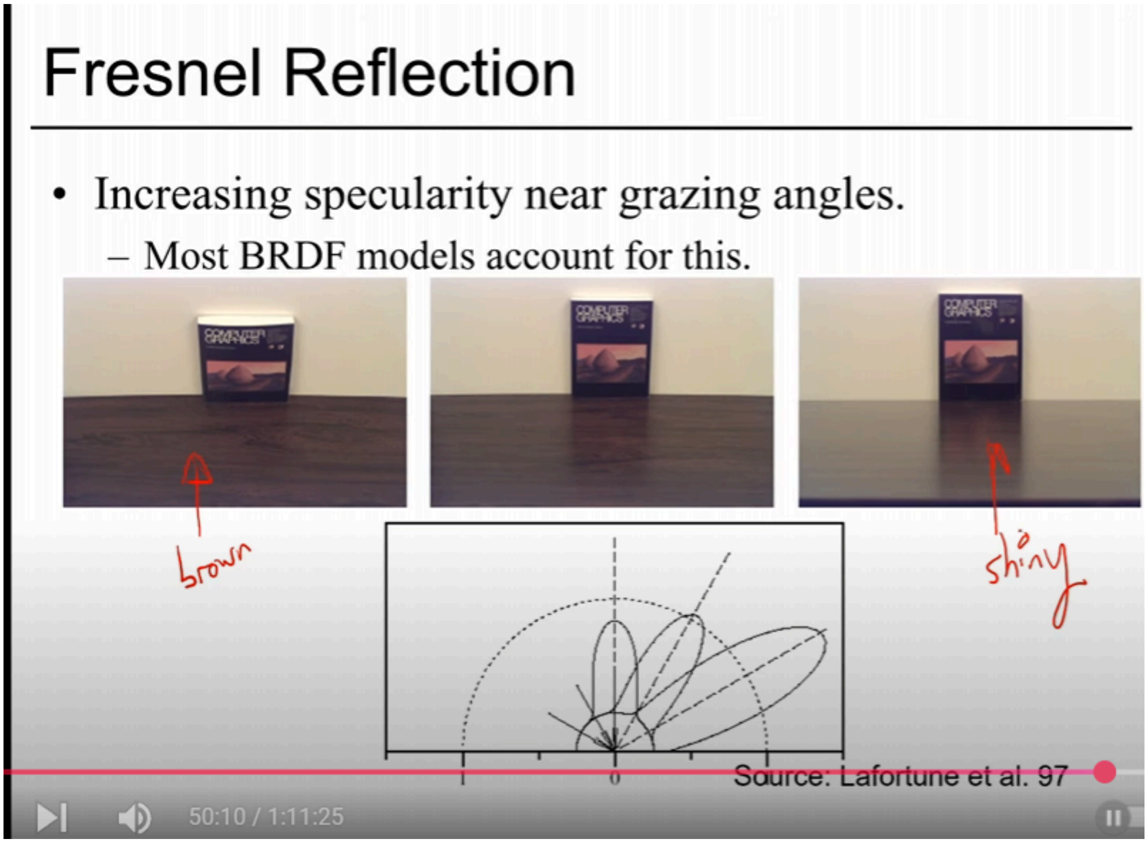


Phong Examples

- The spheres illustrate specular reflections as the direction of the light source and the exponent q (amount of shininess) is varied.

$$L_o = \left[k_a + k_d\,(\boldsymbol{n}\cdot\boldsymbol{l}) + k_s\,(\boldsymbol{v}\cdot\boldsymbol{r})^q\right]\frac{L_i}{r^2}$$

- Fresnel Reflection

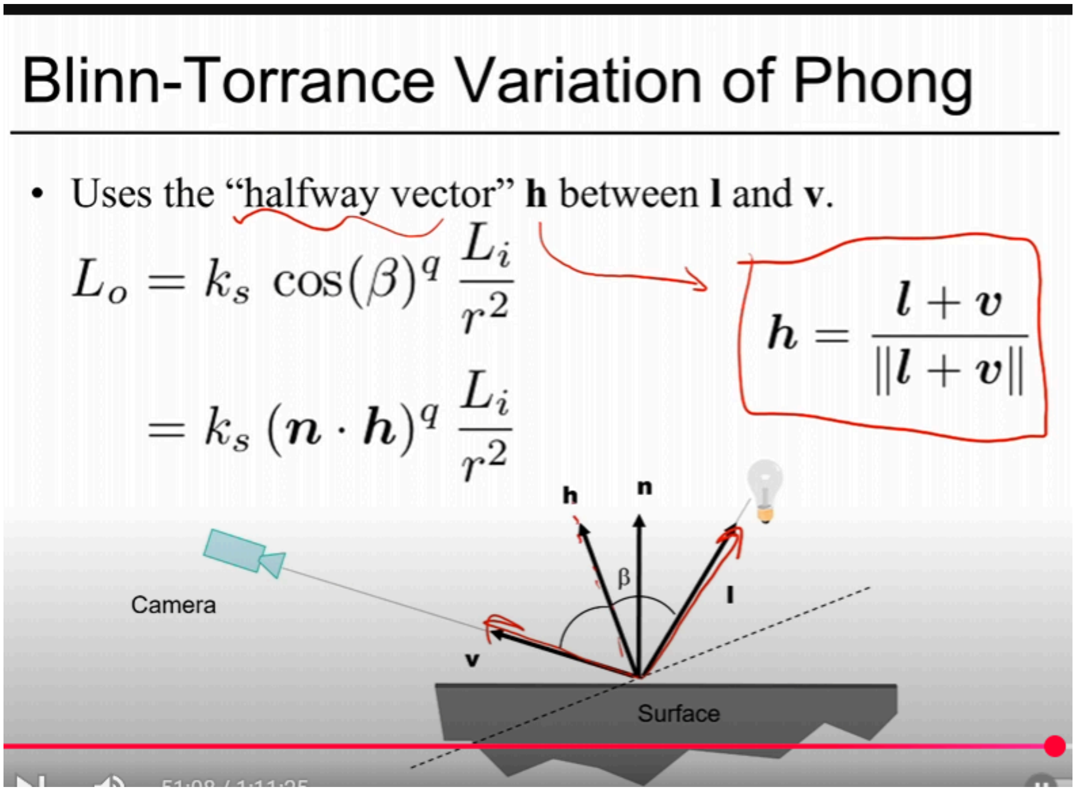


Fresnel Reflection

- Increasing specularity near grazing angles.
  - Most BRDF models account for this.

Source: Lafortune et al. 97

- Blinn-Torrance Half Vector Lobe that support fresnel relfection



Blinn-Torrance Variation of Phong

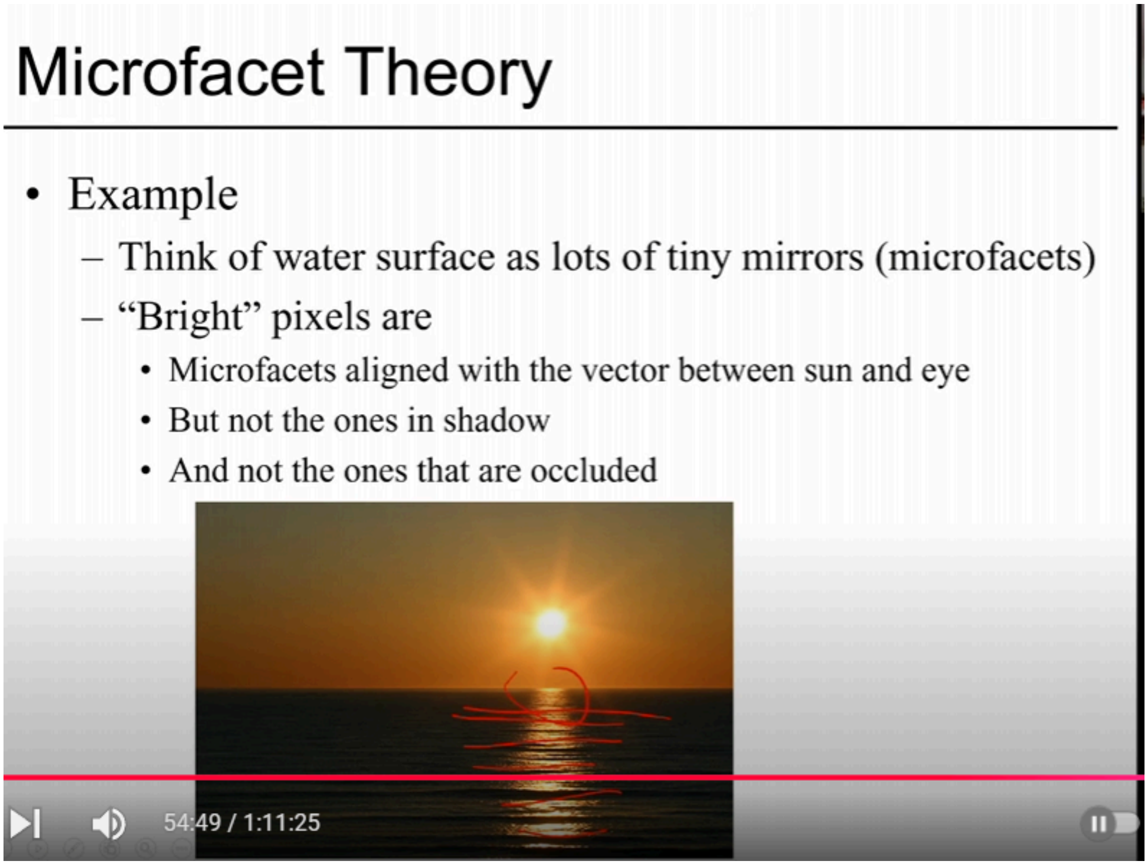
- Uses the “halfway vector” **h** between **l** and **v**.

$$L_o = k_s \cos(\beta)^q \frac{L_i}{r^2}$$

$$= k_s \, (\boldsymbol{n} \cdot \boldsymbol{h})^q \frac{L_i}{r^2}$$

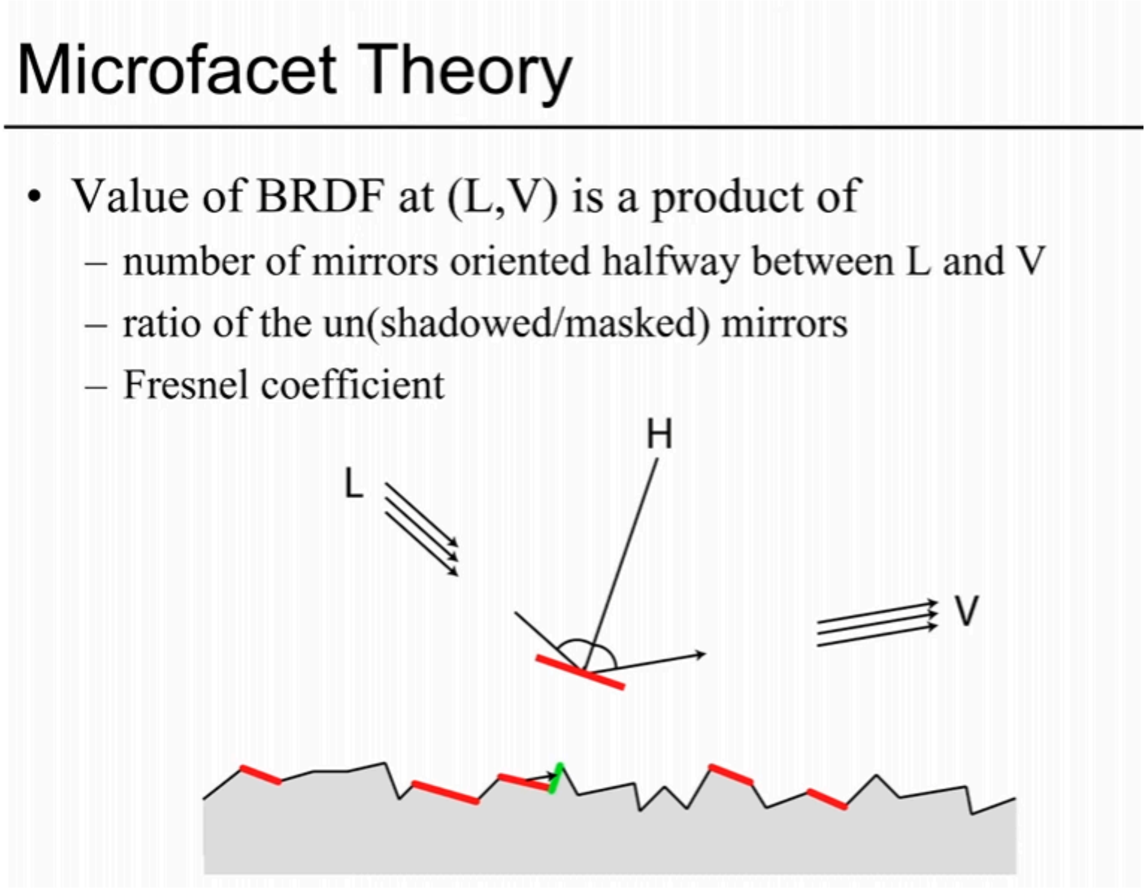
$$\boxed{\boldsymbol{h} = \frac{\boldsymbol{l} + \boldsymbol{v}}{\|\boldsymbol{l} + \boldsymbol{v}\|}}$$

- Microfacet



Microfacet Theory

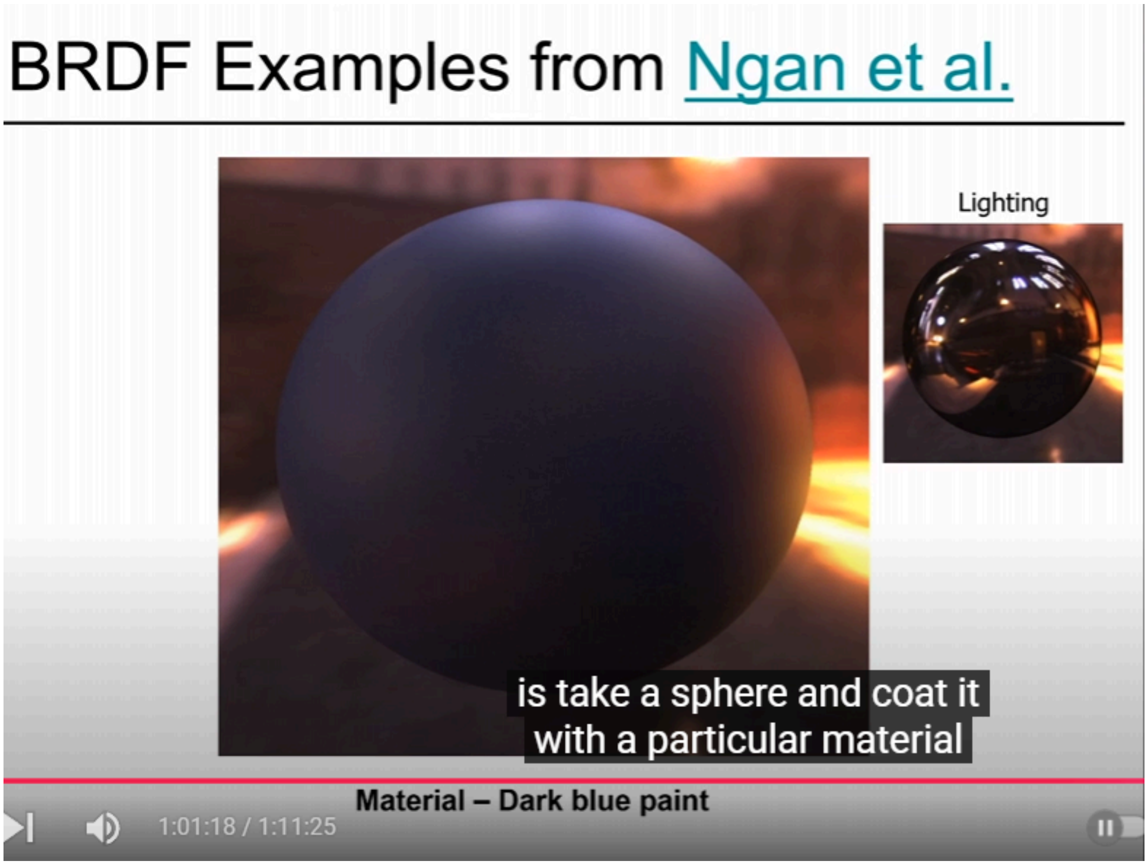
- Example
  - Think of water surface as lots of tiny mirrors (microfacets)
  - “Bright” pixels are
    - Microfacets aligned with the vector between sun and eye
    - But not the ones in shadow
    - And not the ones that are occluded



Microfacet Theory

- Value of BRDF at (L,V) is a product of
  - number of mirrors oriented halfway between L and V
  - ratio of the un(shadowed/masked) mirrors
  - Fresnel coefficient

- Other BRDF Example



BRDF Examples from Ngan et al.

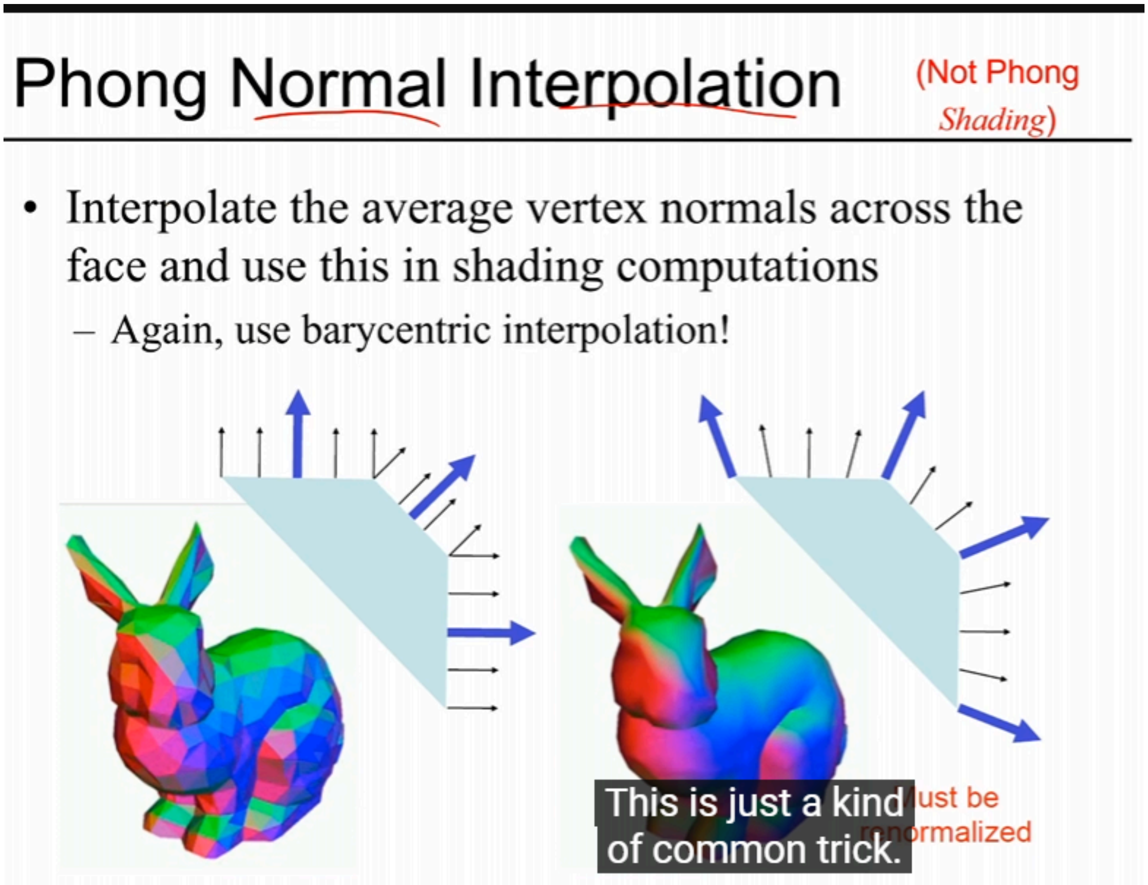
Lighting

is take a sphere and coat it
with a particular material

Material – Dark blue paint

1:01:18 / 1:11:25

    - Phong Normal Interpolation



Phong Normal Interpolation

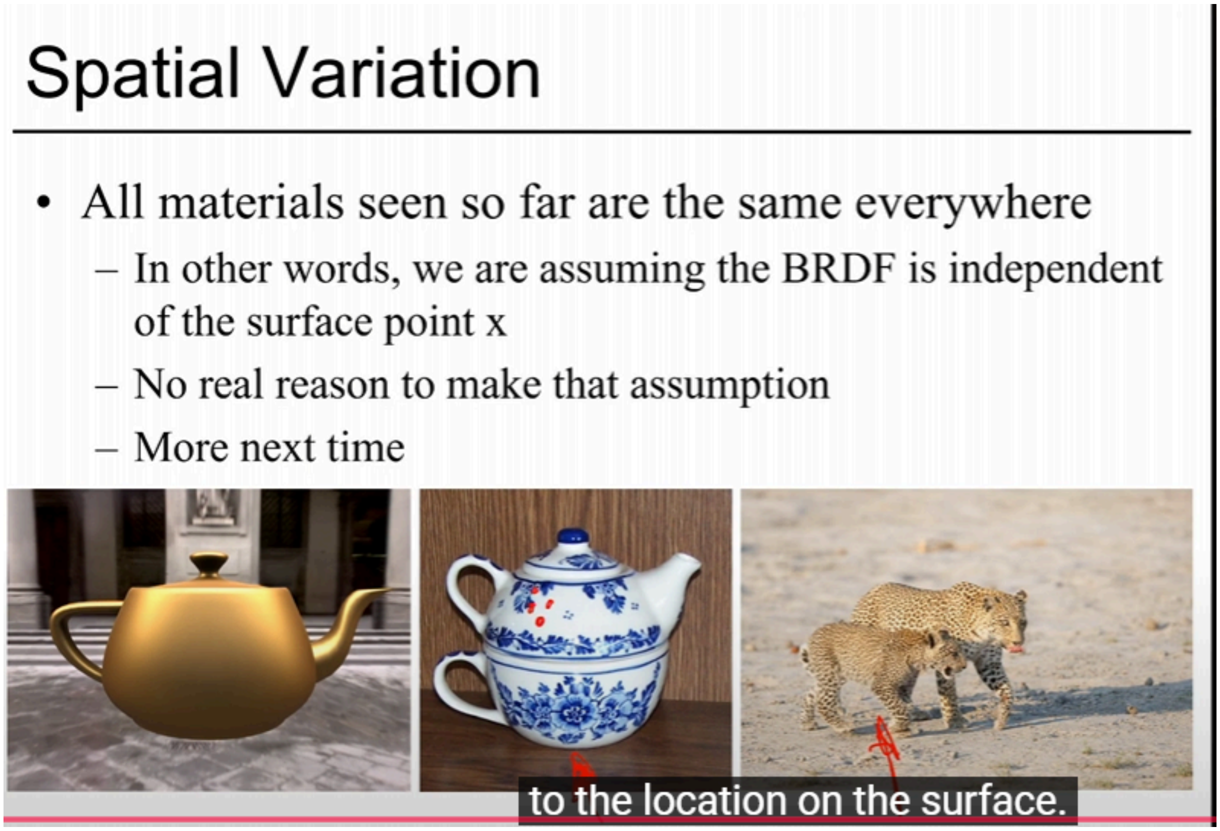
(Not Phong *Shading*)

- Interpolate the average vertex normals across the face and use this in shading computations
  - Again, use barycentric interpolation!

ust be renormalized

This is just a kind of common trick.

  - Spatial Variation



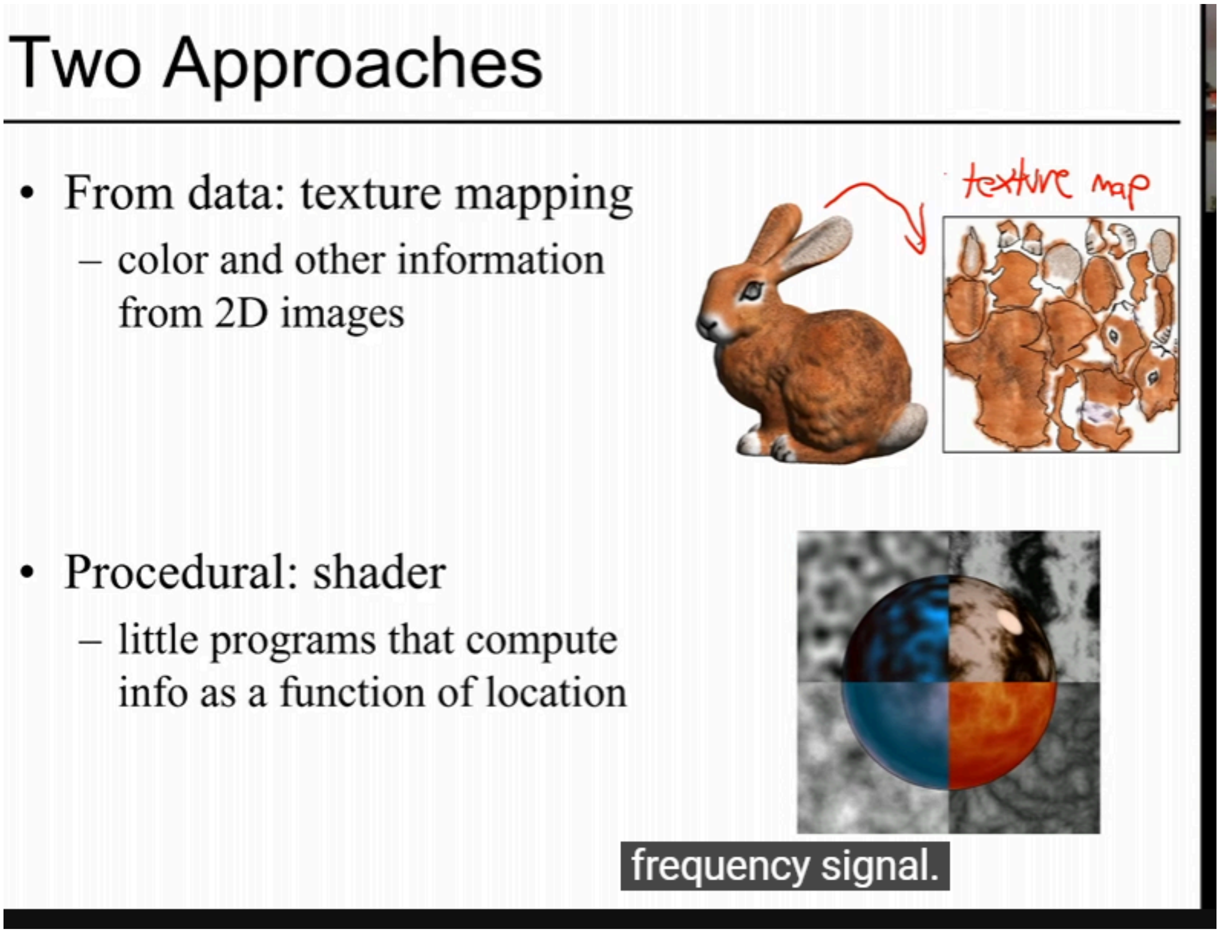
Spatial Variation

- All materials seen so far are the same everywhere
  - In other words, we are assuming the BRDF is independent of the surface point x
  - No real reason to make that assumption
  - More next time

to the location on the surface.

- **L14: Textures, parameterization, shaders, Perlin noise**

- Spatial Variation

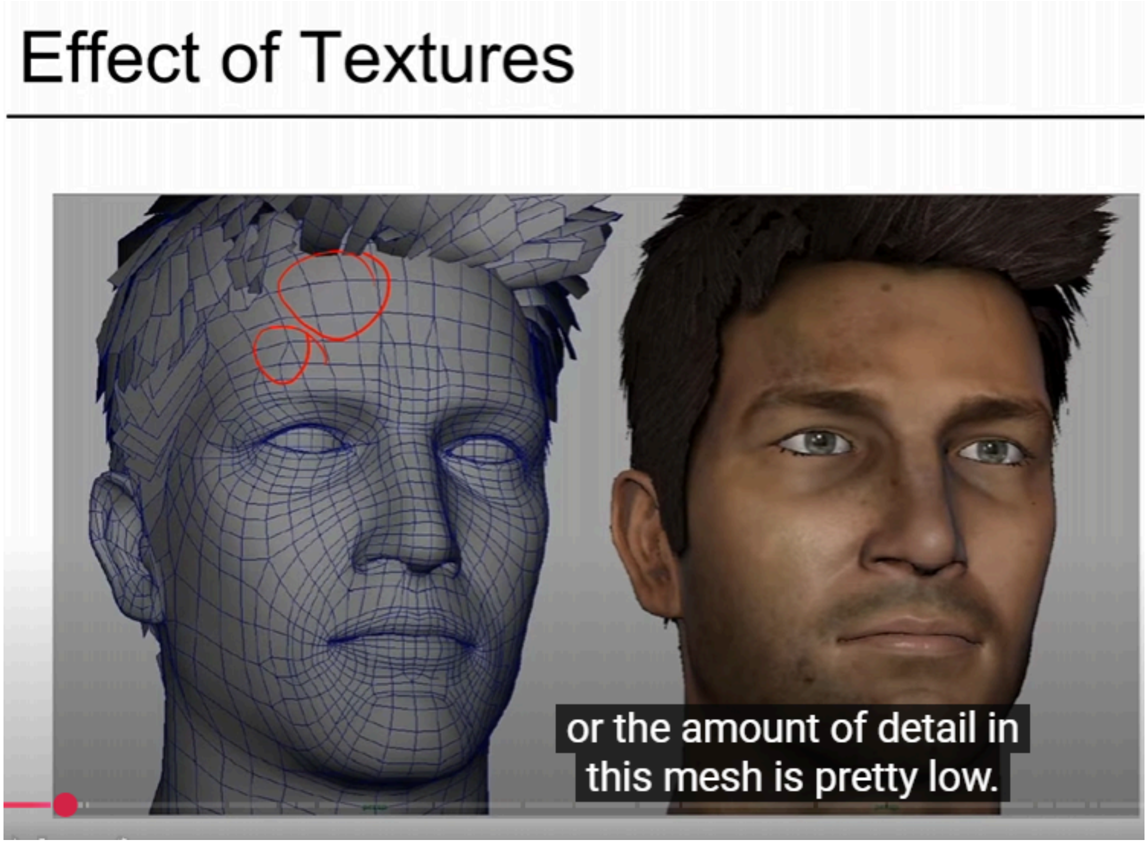


Two Approaches

- From data: texture mapping
  - color and other information from 2D images
- Procedural: shader
  - little programs that compute info as a function of location

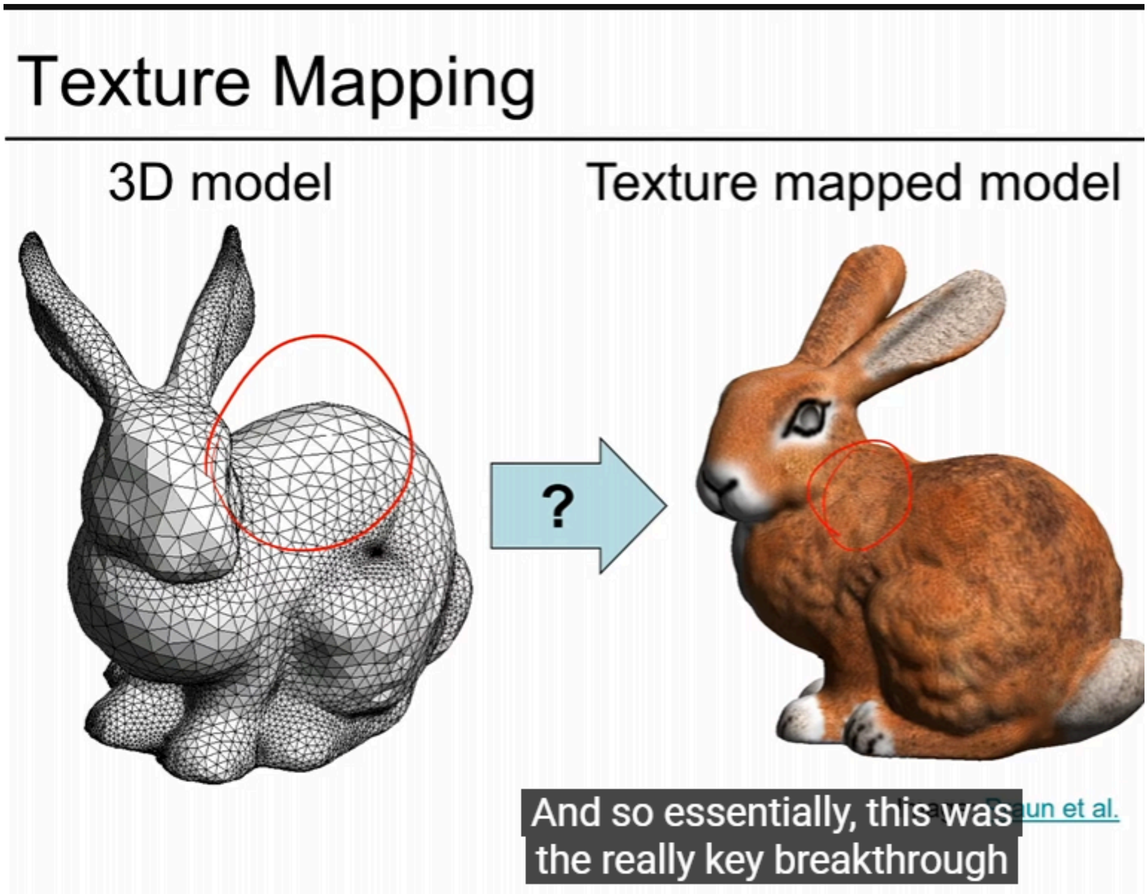
frequency signal.

  - Texture Mapping



Effect of Textures

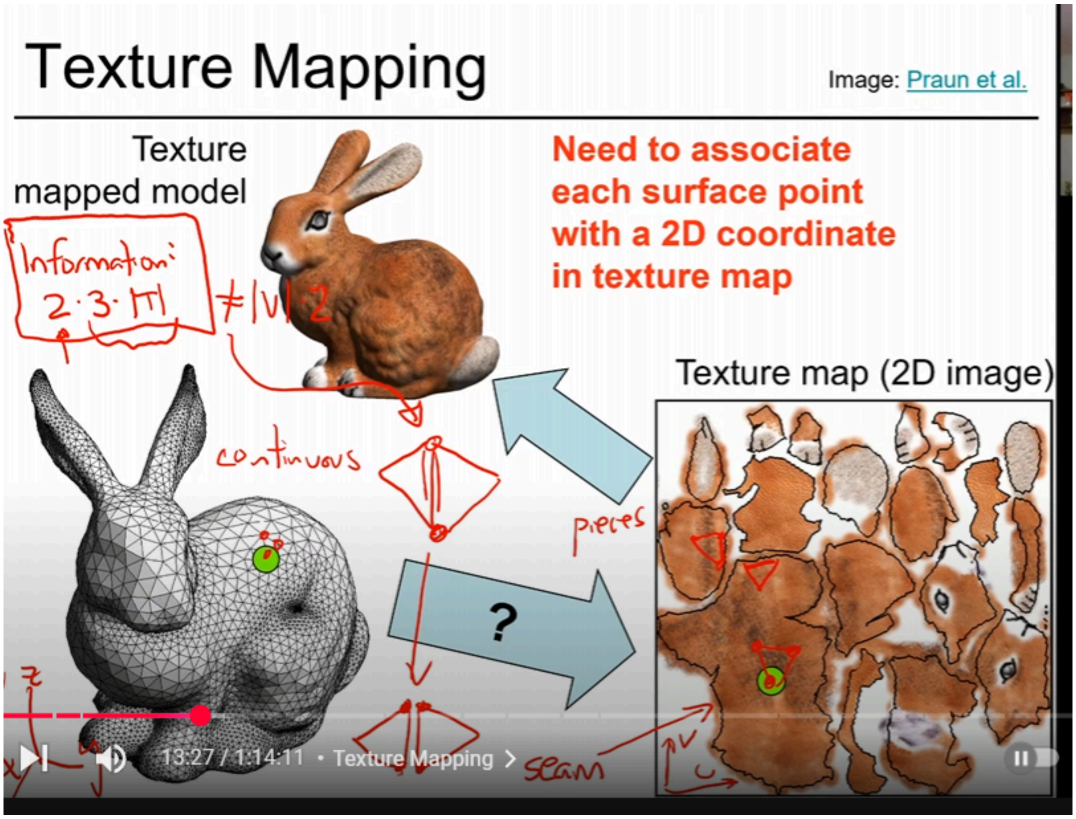
or the amount of detail in this mesh is pretty low.



Texture Mapping

And so essentially, this was the really key breakthrough

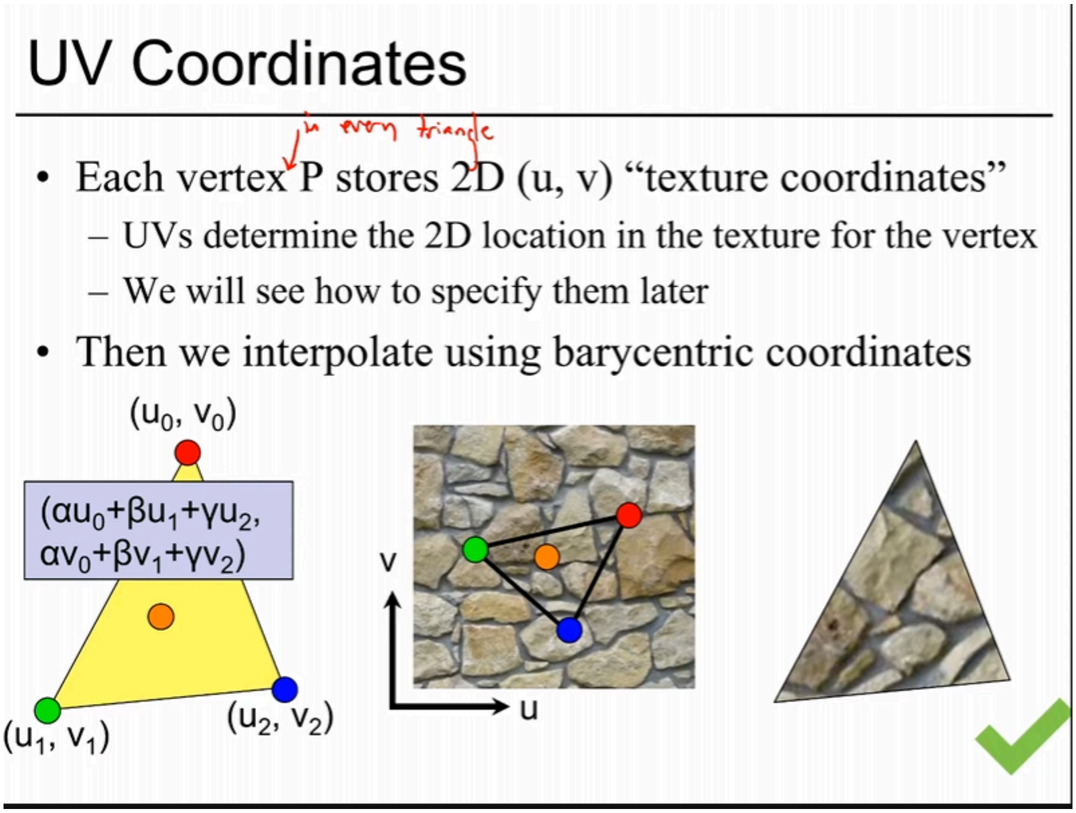
- UV Coordinate



Texture Mapping

Image: Praun et al.

Need to associate each surface point with a 2D coordinate in texture map



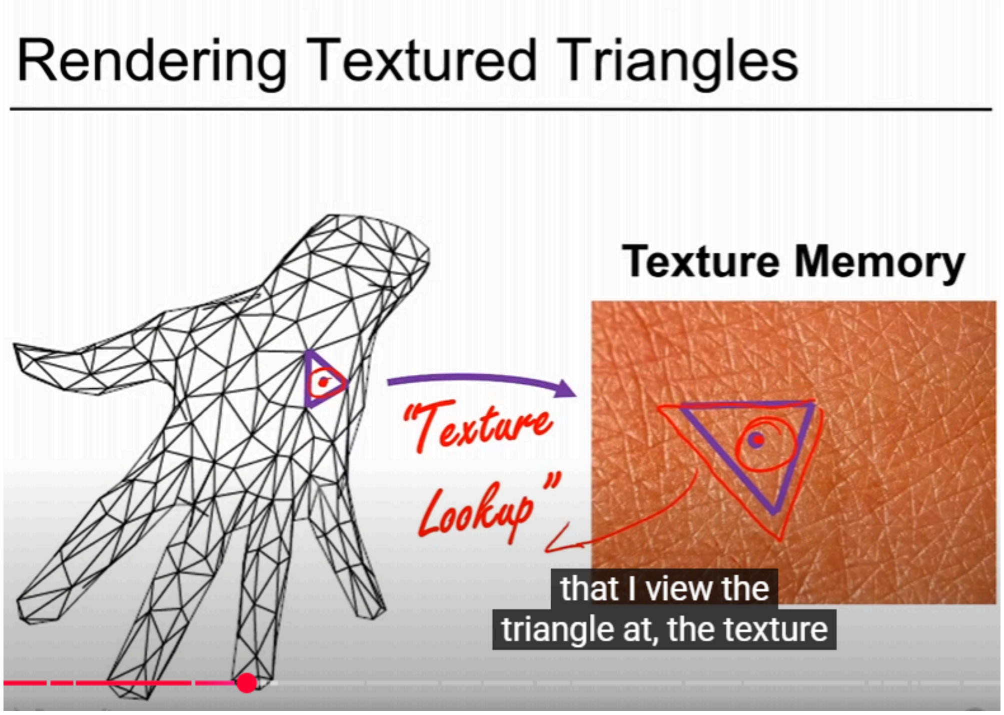
UV Coordinates

in every triangle

- Each vertex P stores 2D (u, v) "texture coordinates"
  - UVs determine the 2D location in the texture for the vertex
  - We will see how to specify them later
- Then we interpolate using barycentric coordinates

(u_0, v_0)
$(\alpha u_0+\beta u_1+\gamma u_2,$
$\alpha v_0+\beta v_1+\gamma v_2)$
(u_1, v_1)
(u_2, v_2)
v
u
✓

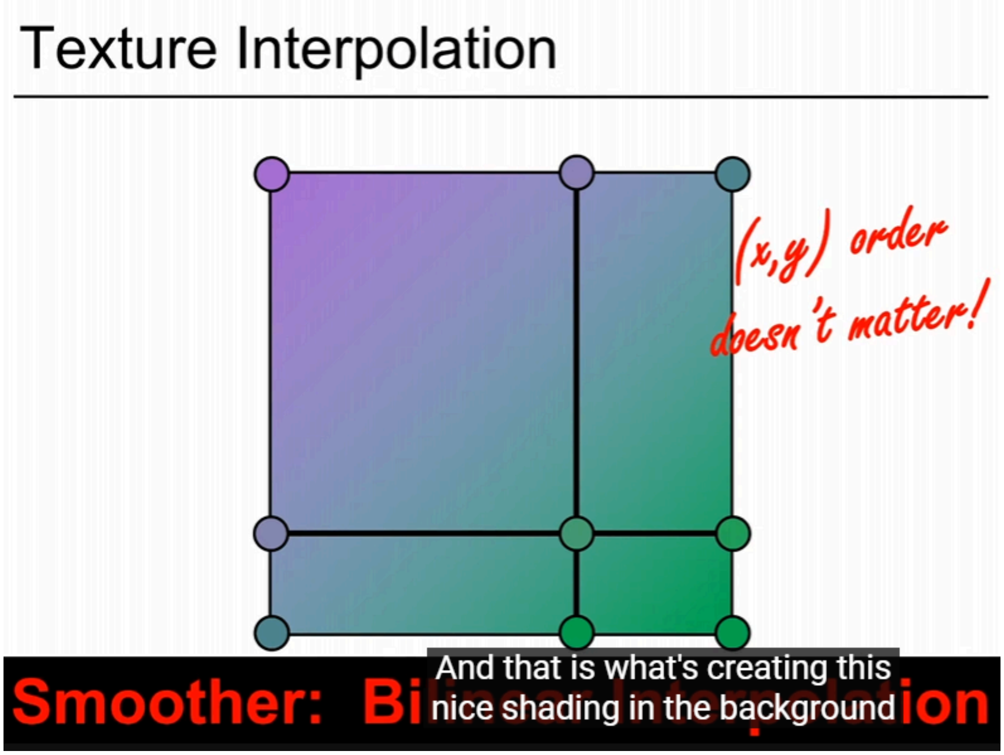
- Rendering Textured Triangles (Texture Lookup)



Rendering Textured Triangles

that I view the
triangle at, the texture

- Texture Interpolation



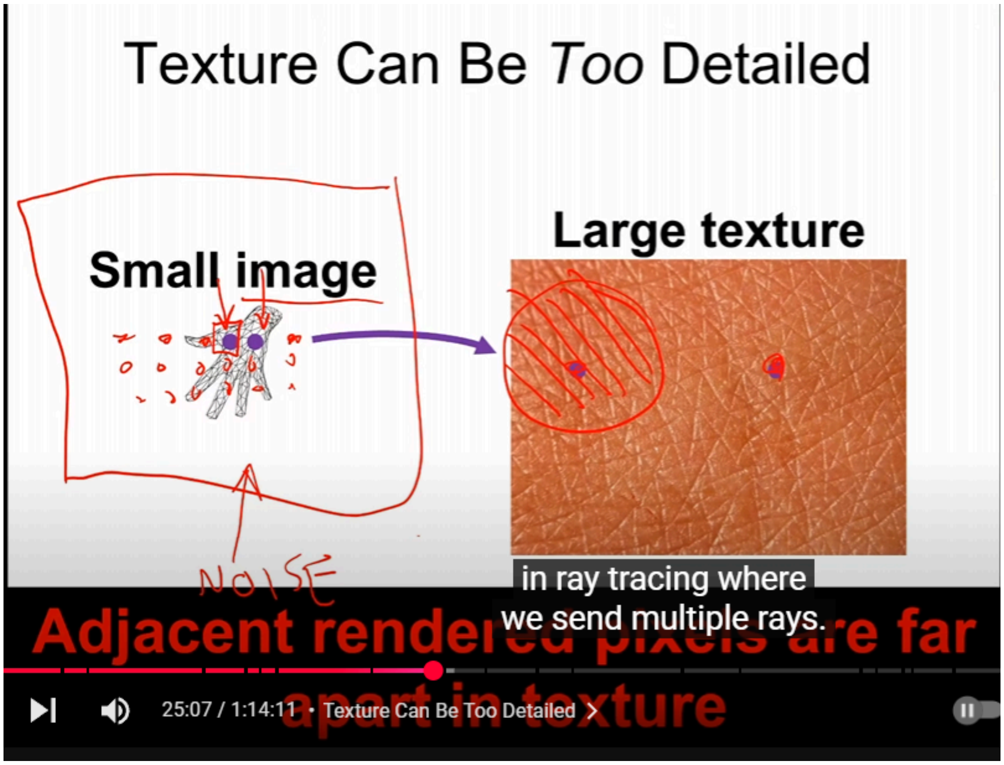
Texture Interpolation

(x,y) order doesn't matter!

Smoother: Bilinear Interpolation

And that is what's creating this nice shading in the background

- Zoom far away, Pixel color too random



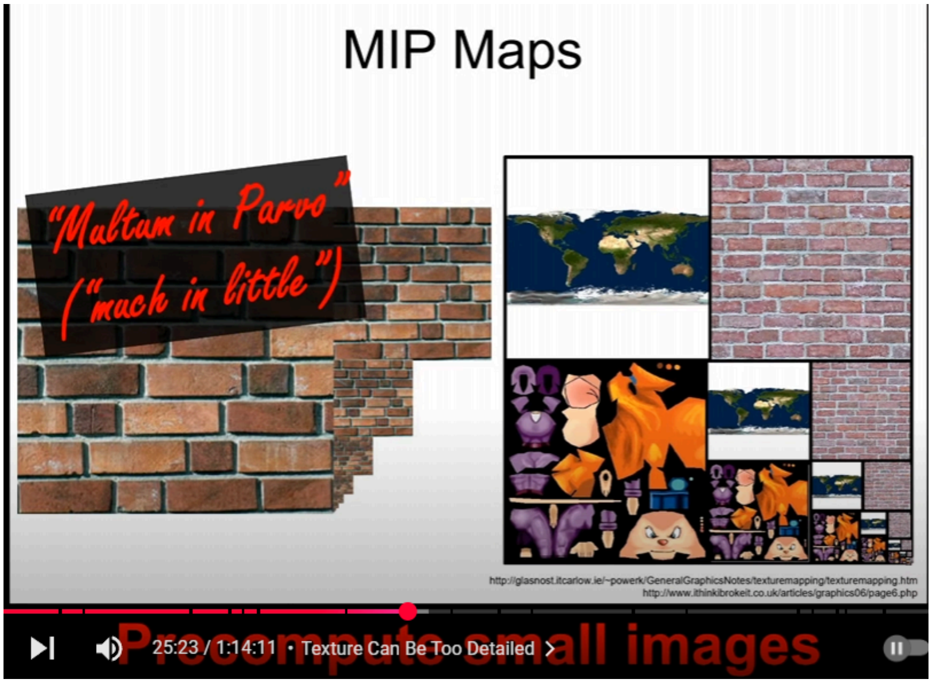
Texture Can Be *Too* Detailed

Adjacent rendered pixels are far apart in texture

in ray tracing where we send multiple rays.

25:07 / 1:14:11 • Texture Can Be Too Detailed

- MIP Maps



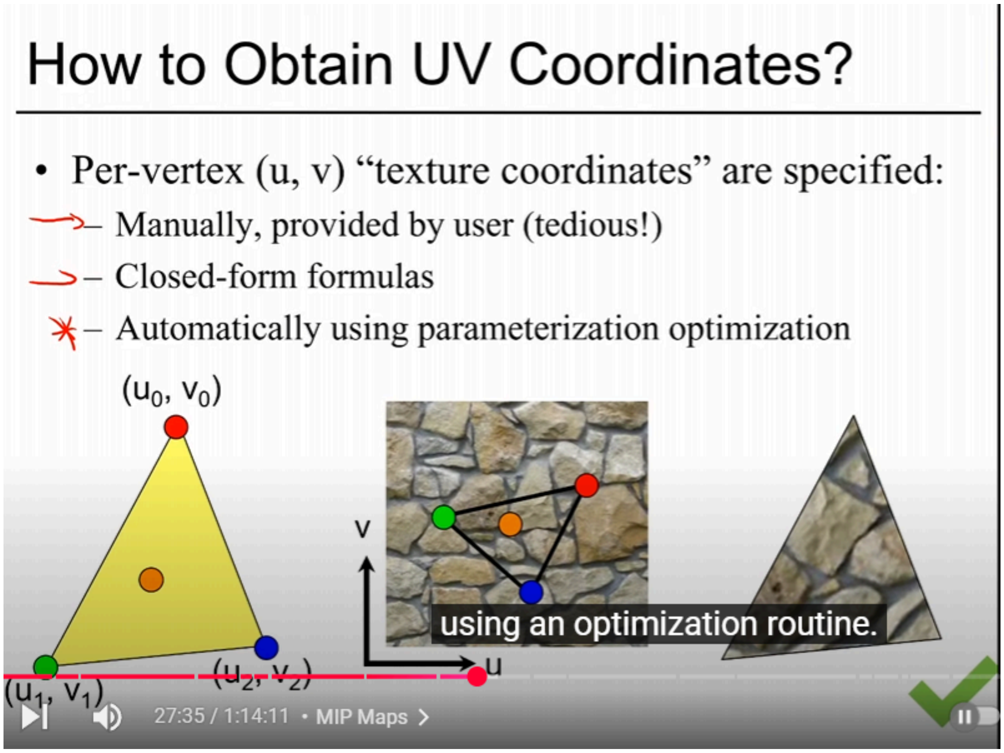
MIP Maps

http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/texturemapping/texturemapping.htm
http://www.ithinkibrokeit.co.uk/articles/graphics06/page6.php

Precompute small images

  - Precompute small images when it is far away

- How to Obtain UV Coordinates

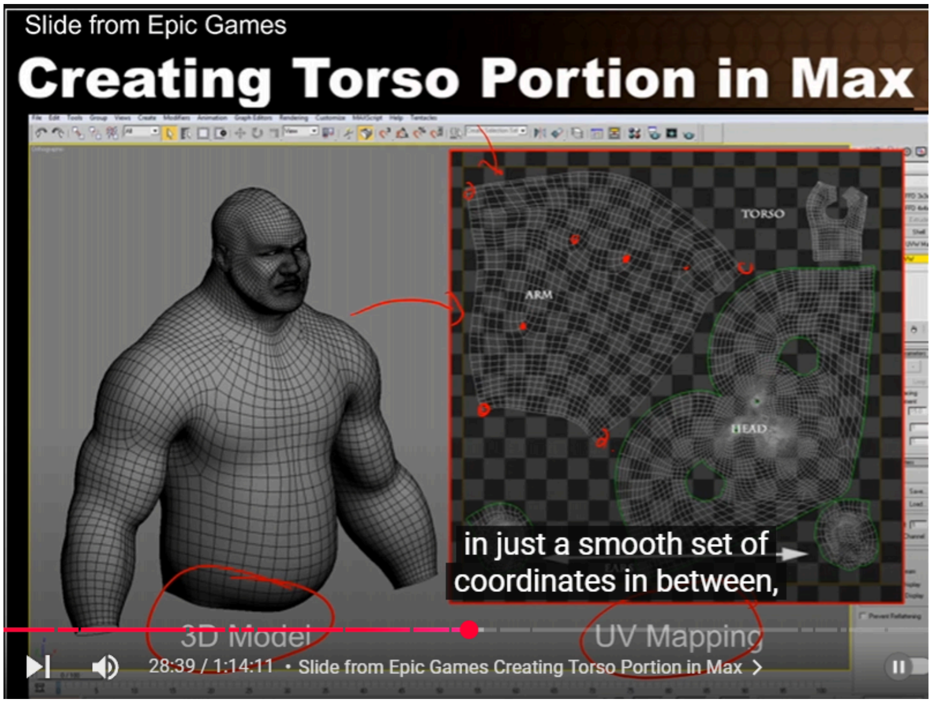


How to Obtain UV Coordinates?

- Per-vertex (u, v) "texture coordinates" are specified:
  - Manually, provided by user (tedious!)
  - Closed-form formulas
  - Automatically using parameterization optimization

using an optimization routine.

- Manual



Slide from Epic Games

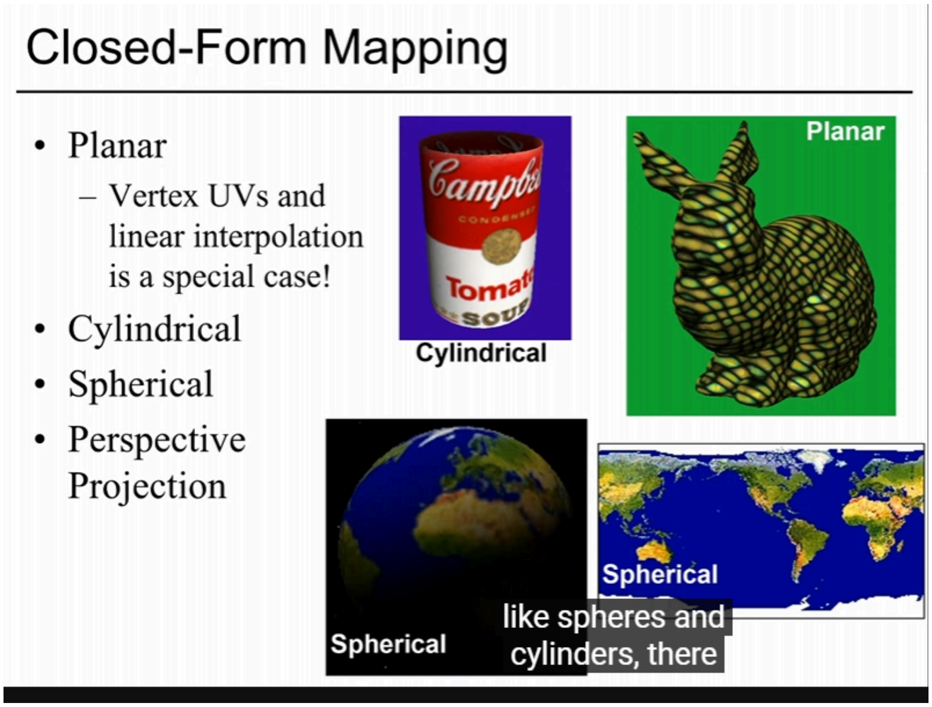
Creating Torso Portion in Max

in just a smooth set of coordinates in between,

28:39 / 1:14:11 • Slide from Epic Games Creating Torso Portion in Max >

  - Artist design key point in the texture

- Closed-Form Mapping



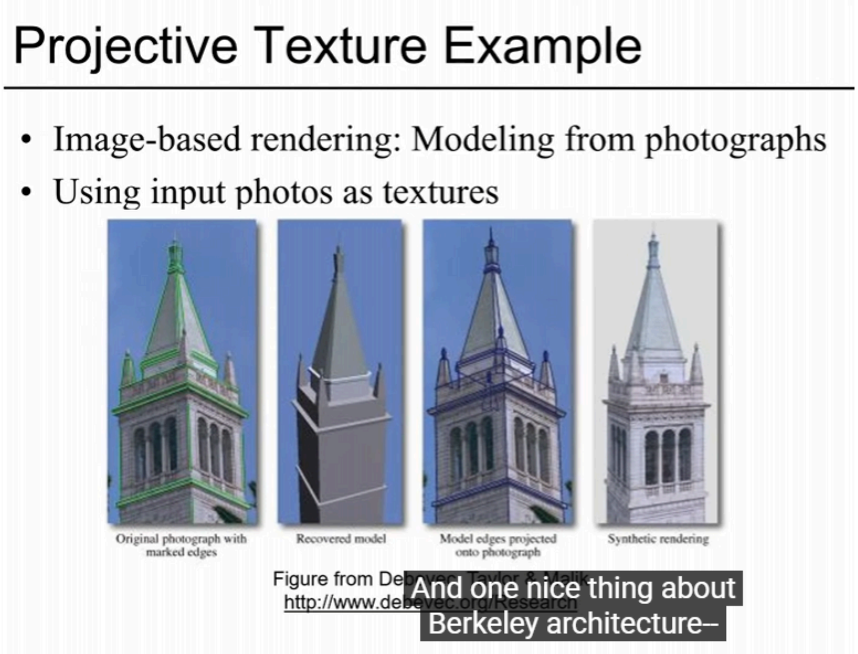
Closed-Form Mapping

- Planar
  - Vertex UVs and linear interpolation is a special case!
- Cylindrical
- Spherical
- Perspective Projection

like spheres and cylinders, there

  - Raycast get height and angle, calculate the shape and get UV

- Projective Mappings



Projective Texture Example

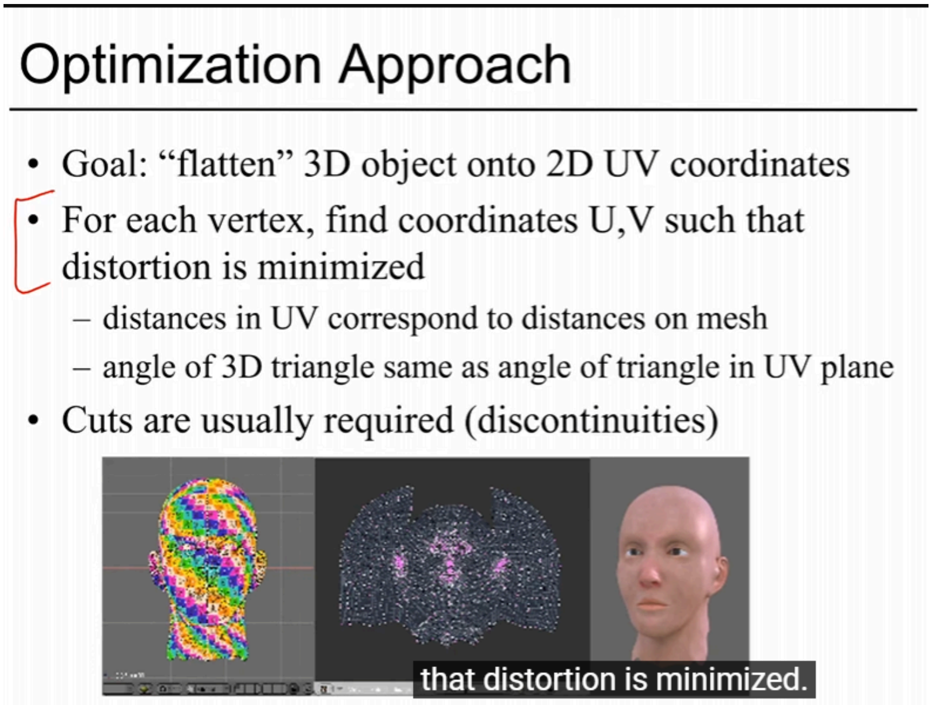
- Image-based rendering: Modeling from photographs
- Using input photos as textures

Original photograph with marked edges
Recovered model
Model edges projected onto photograph
Synthetic rendering

Figure from Debevec, Taylor & Malik
http://www.debevec.org/Research

And one nice thing about Berkeley architecture–

- Optimization Approach

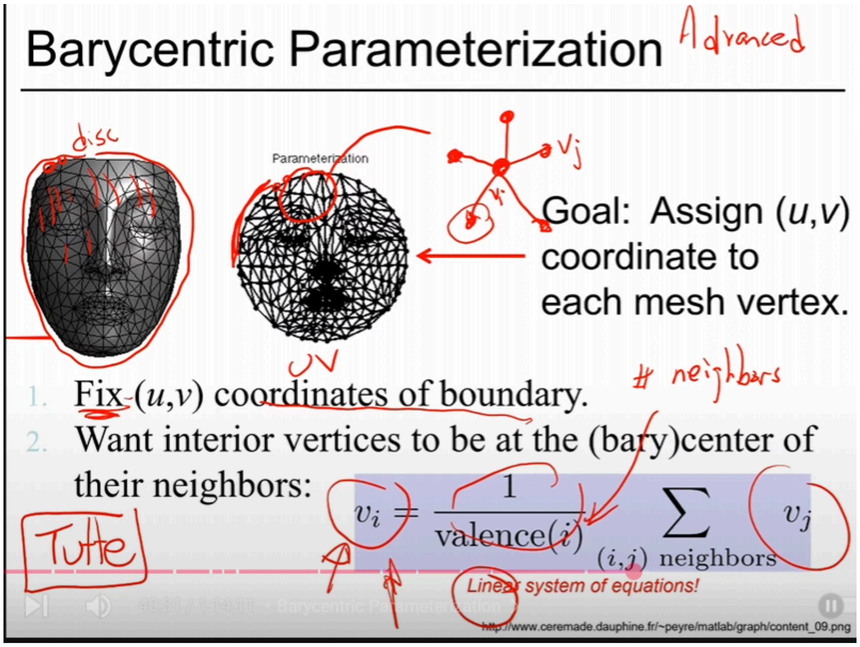


Optimization Approach

- Goal: “flatten” 3D object onto 2D UV coordinates
- For each vertex, find coordinates U,V such that distortion is minimized
  - distances in UV correspond to distances on mesh
  - angle of 3D triangle same as angle of triangle in UV plane
- Cuts are usually required (discontinuities)

that distortion is minimized.

- Barycentric Parameterization



Barycentric Parameterization

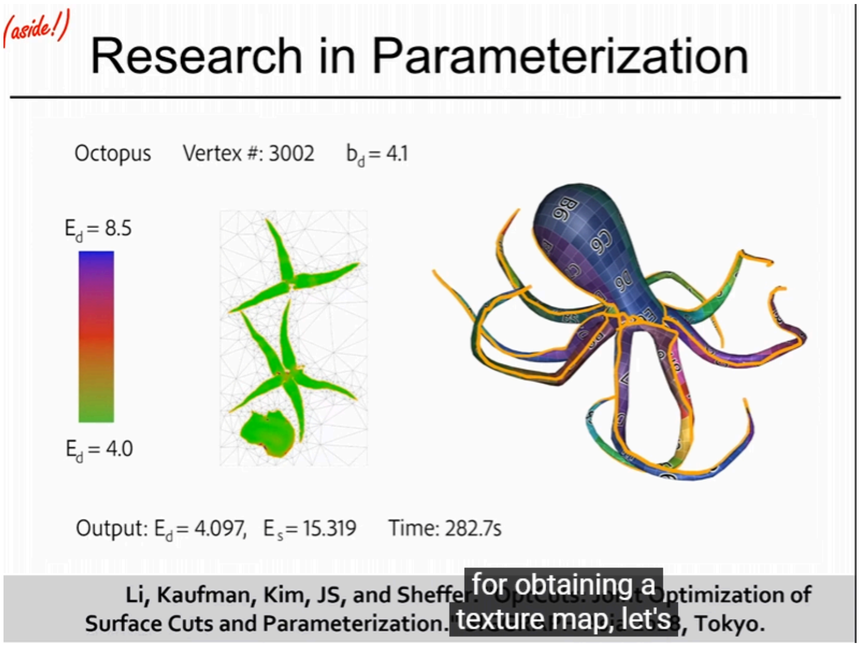
Goal: Assign (*u*,*v*) coordinate to each mesh vertex.

1. Fix (*u*,*v*) coordinates of boundary.
2. Want interior vertices to be at the (bary)center of their neighbors:

$$v_i = \frac{1}{\text{valence}(i)} \sum_{(i,j)\ \text{neighbors}} v_j$$

Linear system of equations!

http://www.ceremade.dauphine.fr/~peyre/matlab/graph/content_09.png



(aside!)

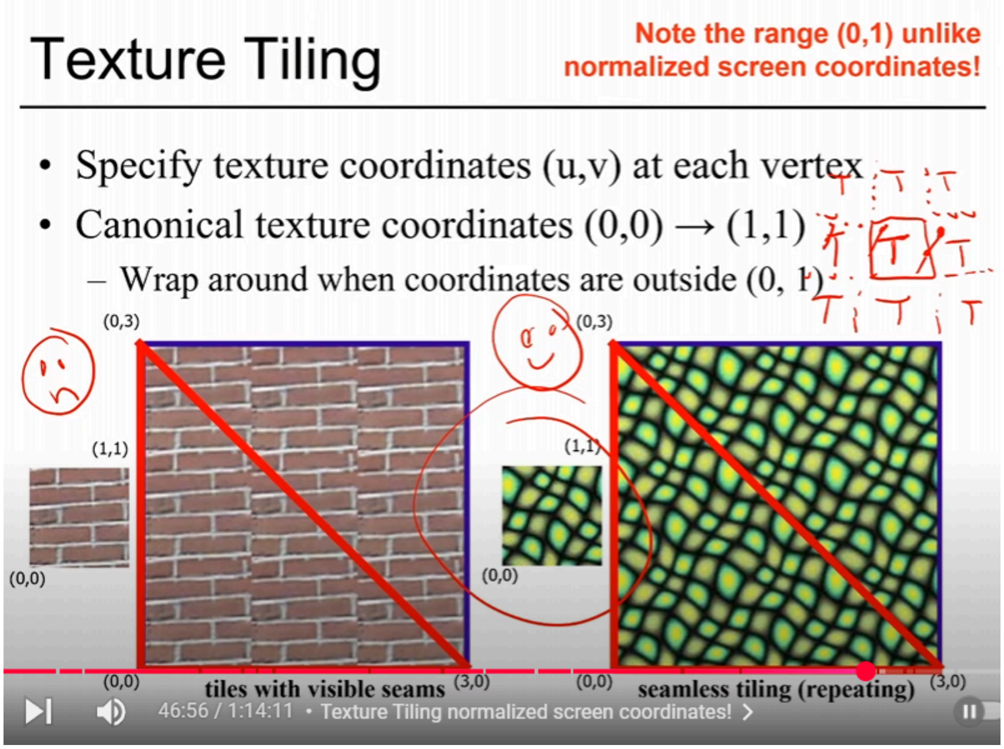
Research in Parameterization

Output: $E_d = 4.097$, $E_s = 15.319$ Time: 282.7s

Li, Kaufman, Kim, JS, and Sheffer. Opt Joint Optimization of Surface Cuts and Parameterization. 3, Tokyo.

for obtaining a texture map, let's

- Texture Tiling



Texture Tiling

Note the range (0,1) unlike normalized screen coordinates!

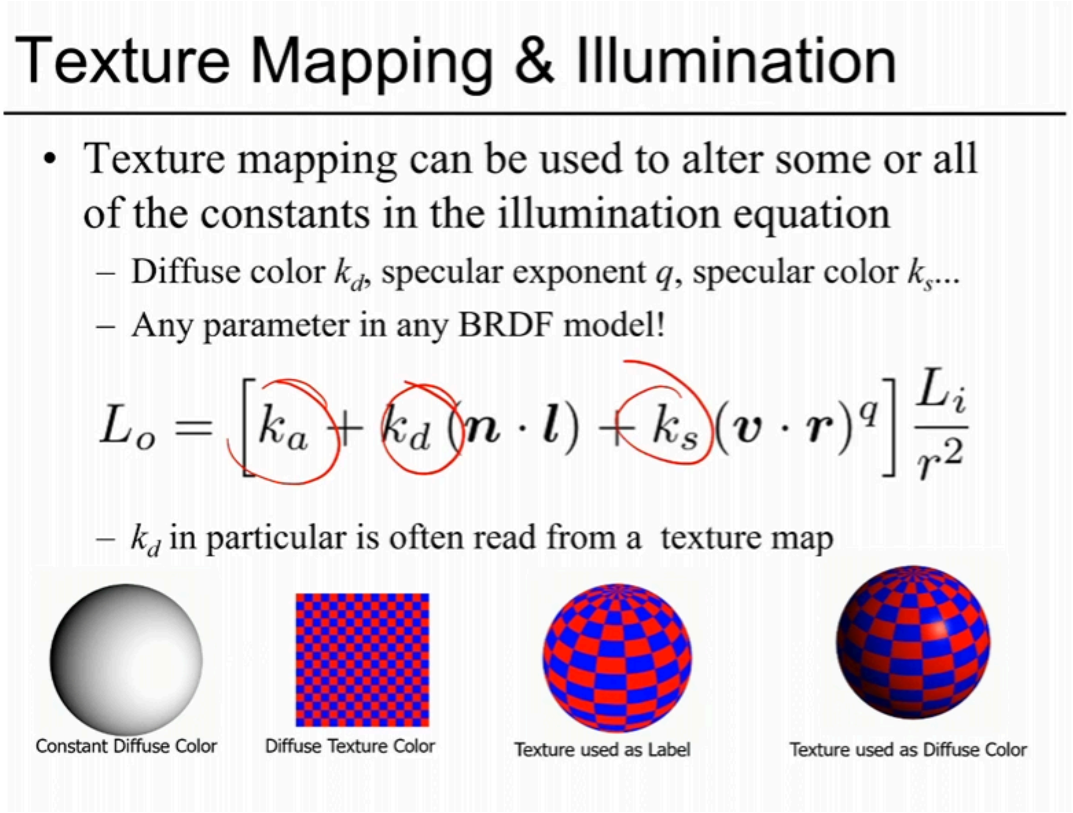
- Specify texture coordinates (u,v) at each vertex
- Canonical texture coordinates (0,0) → (1,1)
  - Wrap around when coordinates are outside (0, 1)

(0,3)
(1,1)
(0,0)
(0,0)
(3,0)
(0,3)
(1,1)
(0,0)
(0,0)
(3,0)
46:56 / 1:14:11 • Texture Tiling normalized screen coordinates! >

tiles with visible seams

seamless tiling (repeating)

- Texture Mapping & Illumination



Texture Mapping & Illumination

- Texture mapping can be used to alter some or all of the constants in the illumination equation
  - Diffuse color k_d, specular exponent q, specular color k_s...
  - Any parameter in any BRDF model!

$$L_o = \left[k_a + k_d (\boldsymbol{n} \cdot \boldsymbol{l}) + k_s (\boldsymbol{v} \cdot \boldsymbol{r})^q \right] \frac{L_i}{r^2}$$

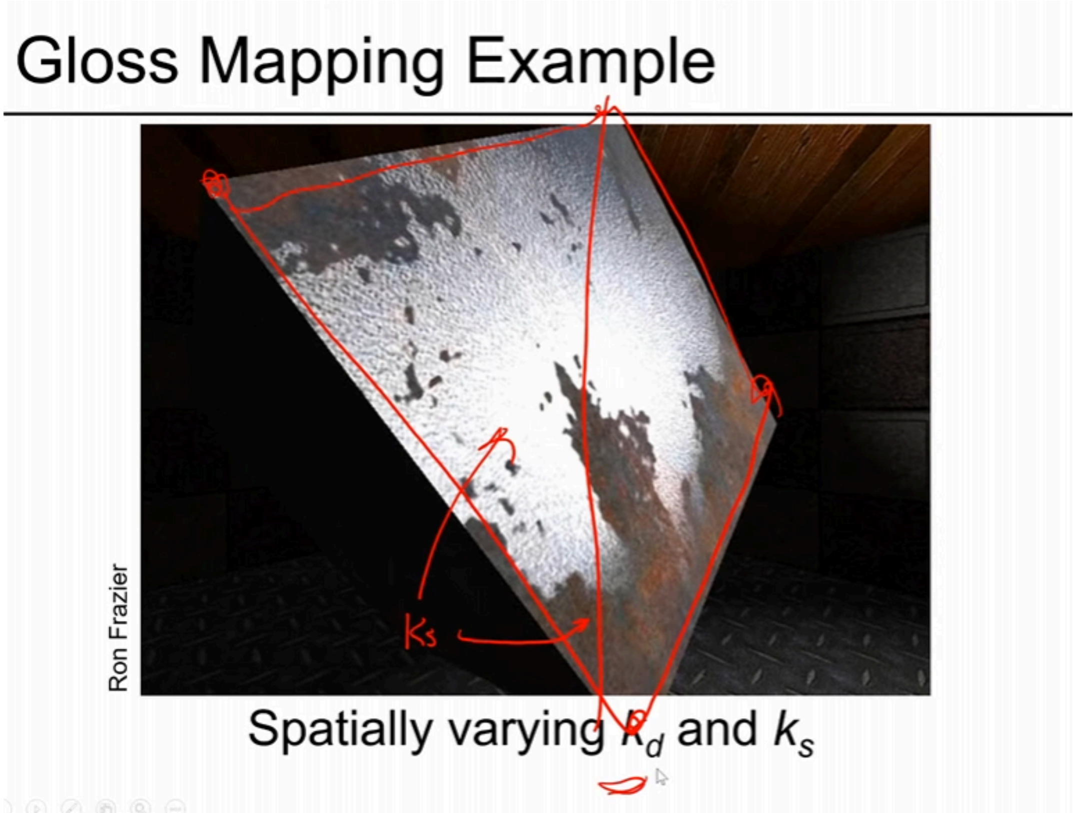
  - k_d in particular is often read from a texture map

Constant Diffuse Color

Diffuse Texture Color

Texture used as Label

Texture used as Diffuse Color

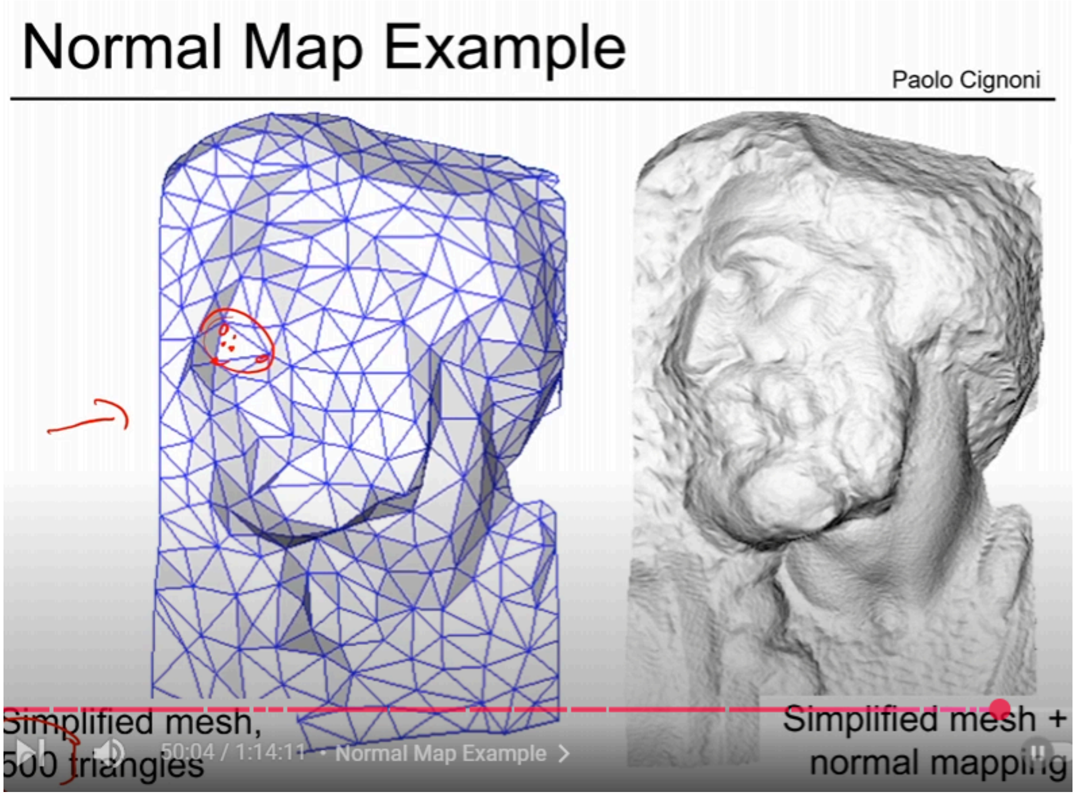


Gloss Mapping Example

Ron Frazier

Spatially varying k_d and k_s

- Normal Mapping

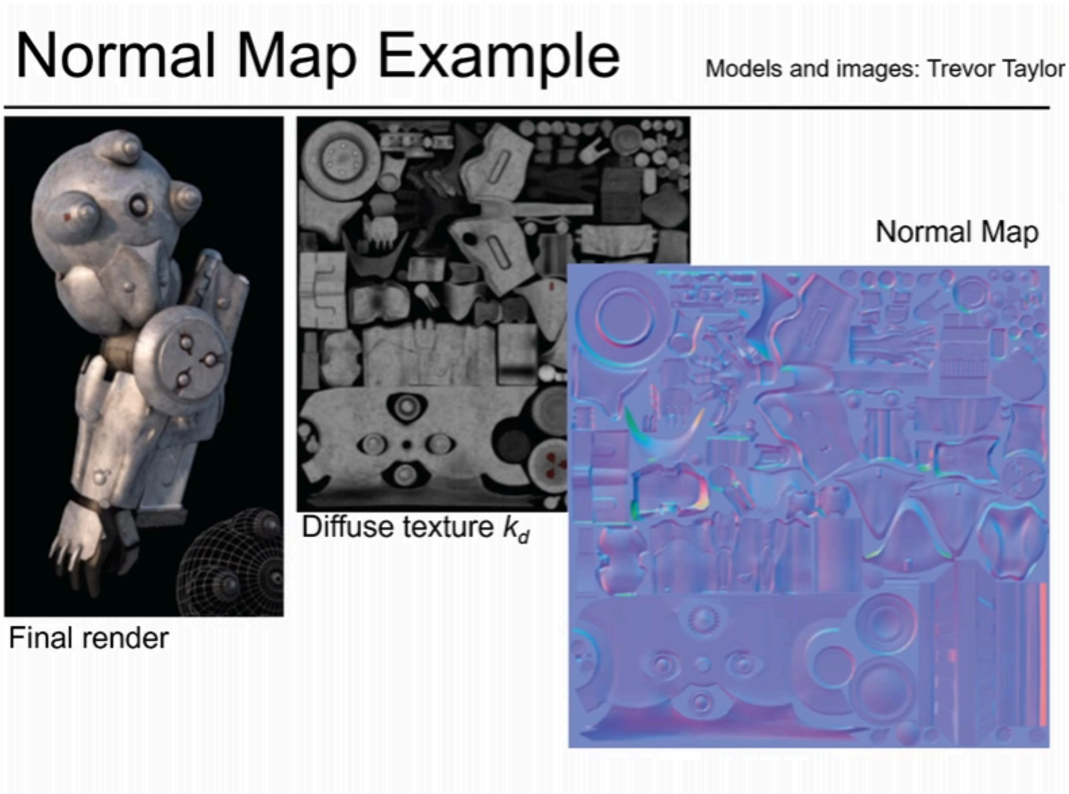


Normal Map Example

Paolo Cignoni

Simplified mesh, 500 triangles

Simplified mesh + normal mapping



Normal Map Example

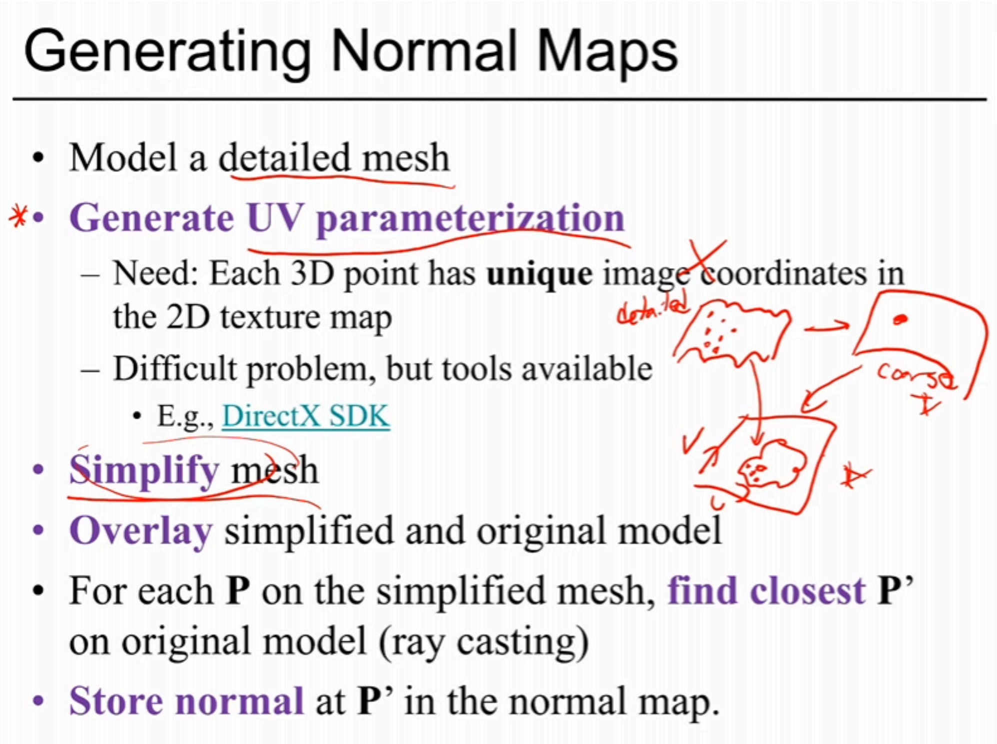
Models and images: Trevor Taylor

Final render

Diffuse texture k_d

Normal Map

- Generating Normal Maps

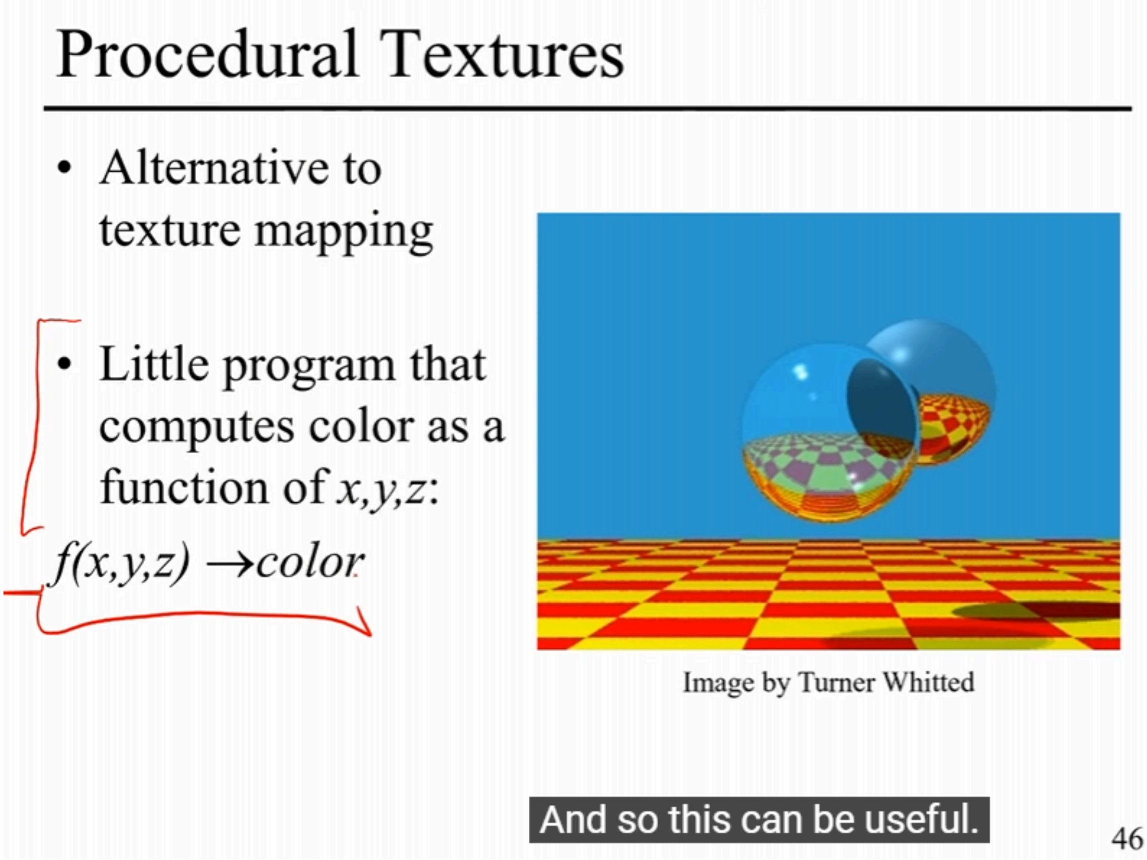


Generating Normal Maps

- Model a detailed mesh
- **Generate UV parameterization**
  - Need: Each 3D point has **unique** image coordinates in the 2D texture map
  - Difficult problem, but tools available
    - E.g., DirectX SDK
- **Simplify** mesh
- **Overlay** simplified and original model
- For each **P** on the simplified mesh, **find closest P'** on original model (ray casting)
- **Store normal** at **P'** in the normal map.

1. Make a detailed mesh
2. Generate UV normal map based on detailed mesh
3. Simplily the mesh
4. Use the simplified mesh with normal map

- Procedural Textures: Shader



Procedural Textures

- Alternative to texture mapping
- Little program that computes color as a function of *x,y,z*:

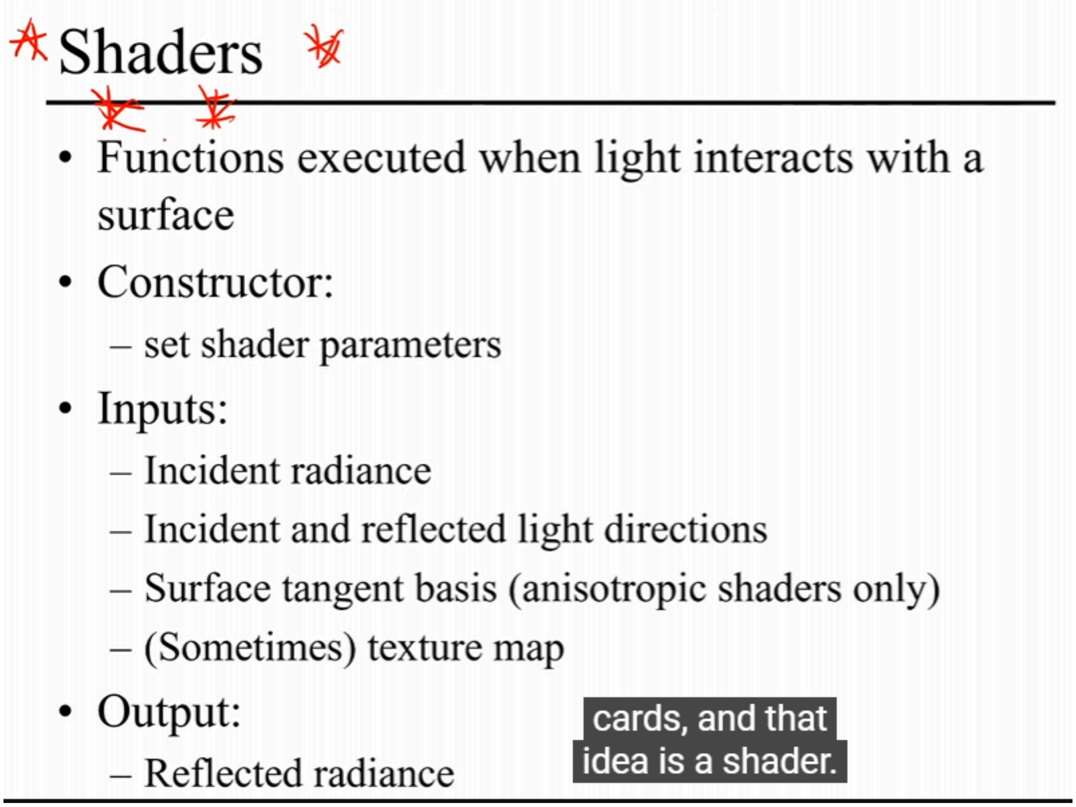
$f(x,y,z) \rightarrow color$

Image by Turner Whitted

And so this can be useful.

46

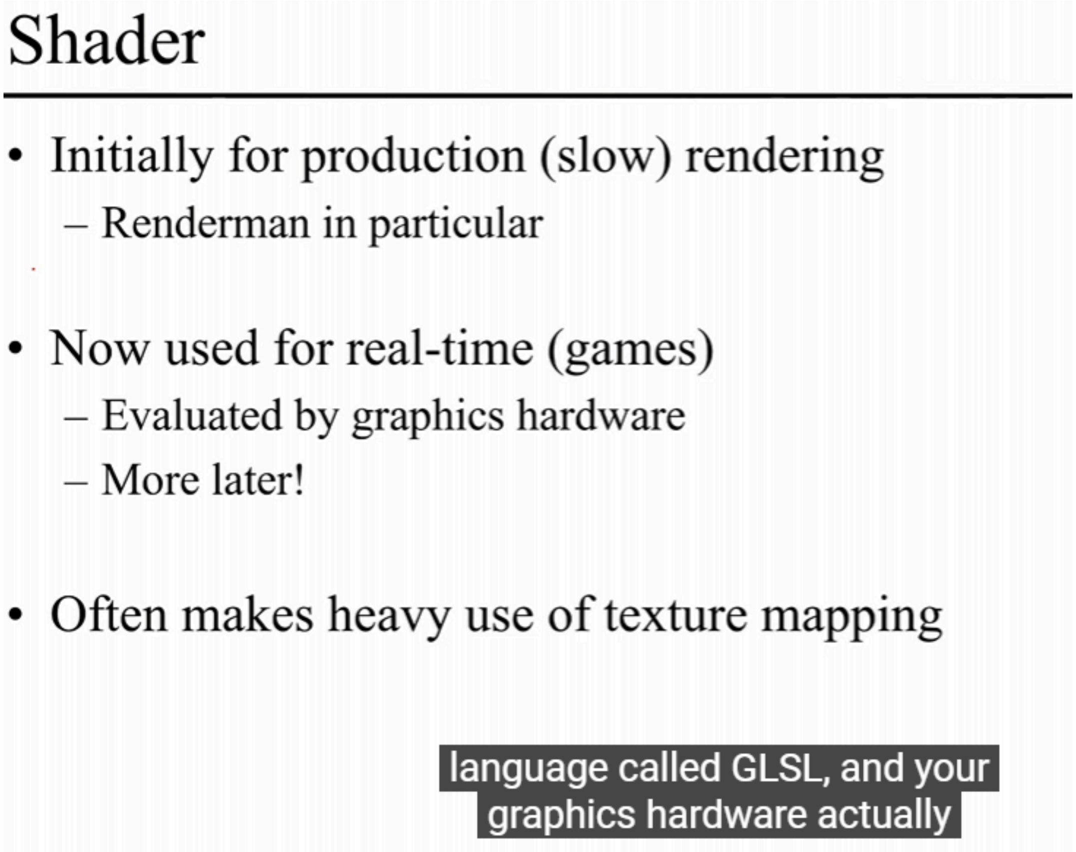
- Shaders



Shaders

- Functions executed when light interacts with a surface
- Constructor:
  - set shader parameters
- Inputs:
  - Incident radiance
  - Incident and reflected light directions
  - Surface tangent basis (anisotropic shaders only)
  - (Sometimes) texture map
- Output:
  - Reflected radiance

cards, and that idea is a shader.

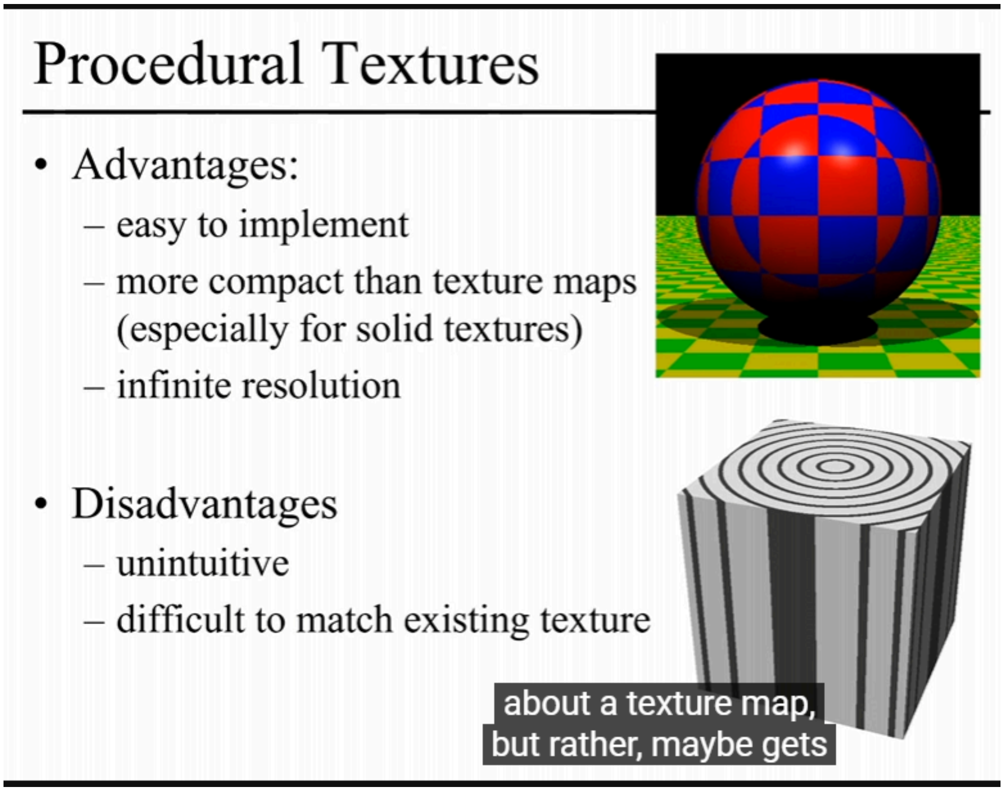


Shader

- Initially for production (slow) rendering
  - Renderman in particular
- Now used for real-time (games)
  - Evaluated by graphics hardware
  - More later!
- Often makes heavy use of texture mapping

language called GLSL, and your
graphics hardware actually

- Pros and Cons

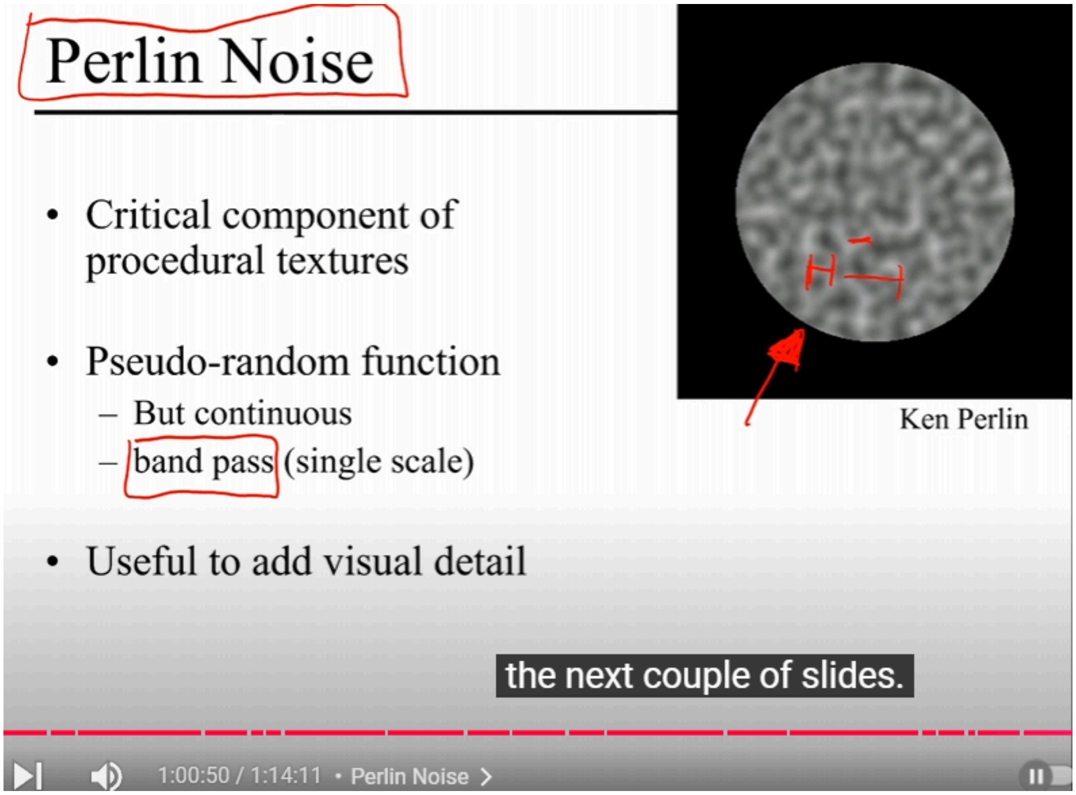


Procedural Textures

- Advantages:
  - easy to implement
  - more compact than texture maps (especially for solid textures)
  - infinite resolution
- Disadvantages
  - unintuitive
  - difficult to match existing texture

about a texture map,
but rather, maybe gets

- Perlin Noise



Perlin Noise

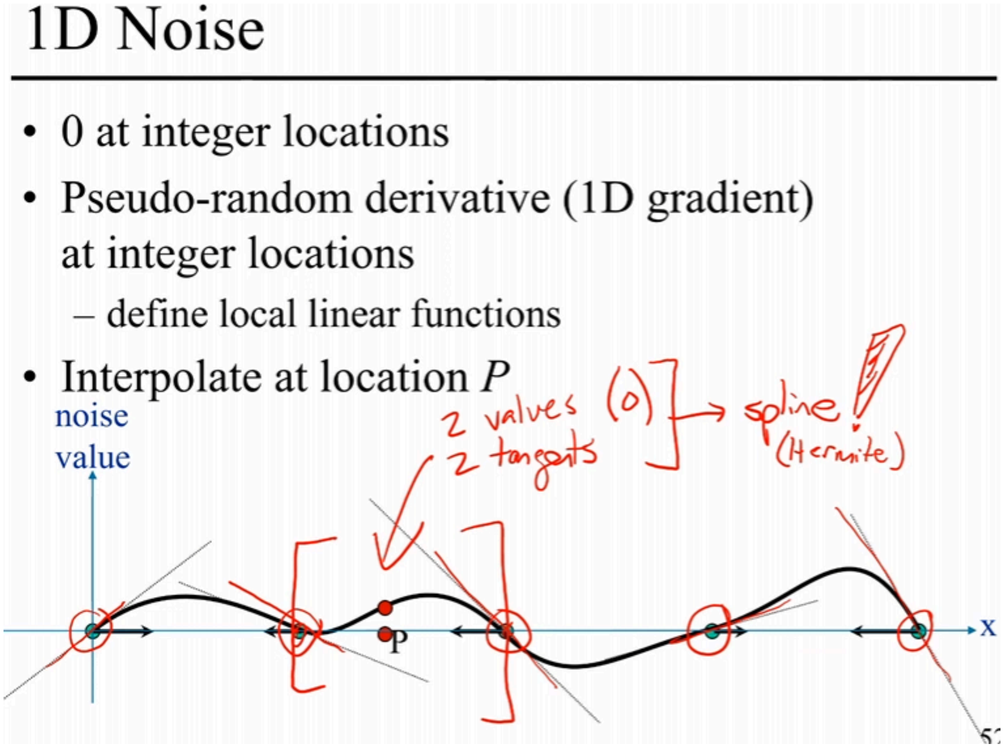
- Critical component of procedural textures
- Pseudo-random function
  - But continuous
  - band pass (single scale)
- Useful to add visual detail

Ken Perlin

the next couple of slides.

  - Requirements
    - Pseudo random
    - For arbitrary dimension
      - 4D is common for animation
    - Smooth at prescribed scale
    - Little memory usage

- 1D Noise

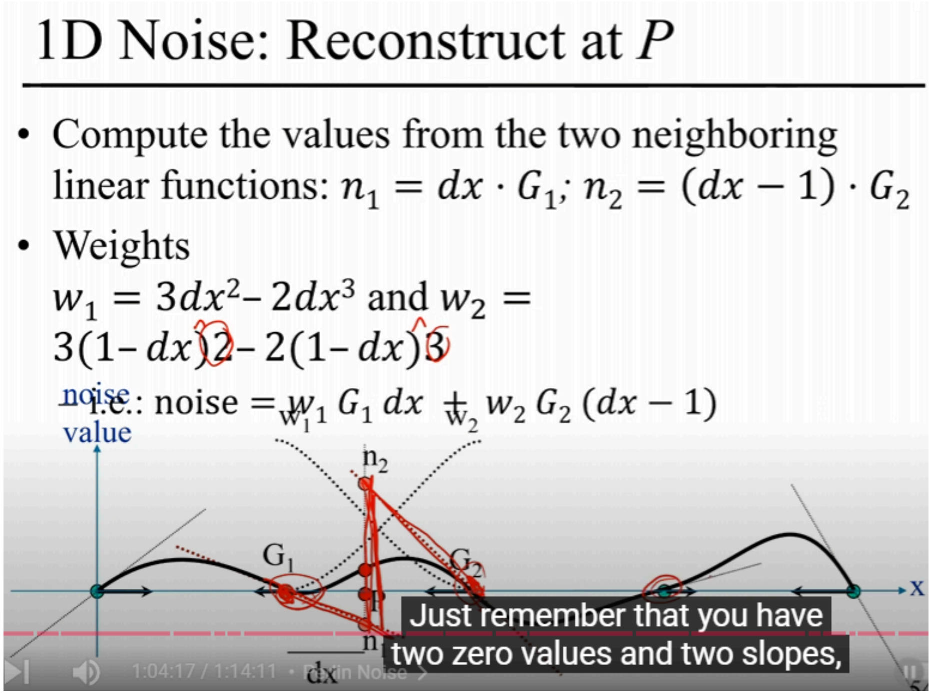


1D Noise

- 0 at integer locations
- Pseudo-random derivative (1D gradient) at integer locations
  - define local linear functions
- Interpolate at location *P*

2 values (0)
2 tangents
→ spline!
(Hermite)

  - Use spline
  - Reconstruct at P



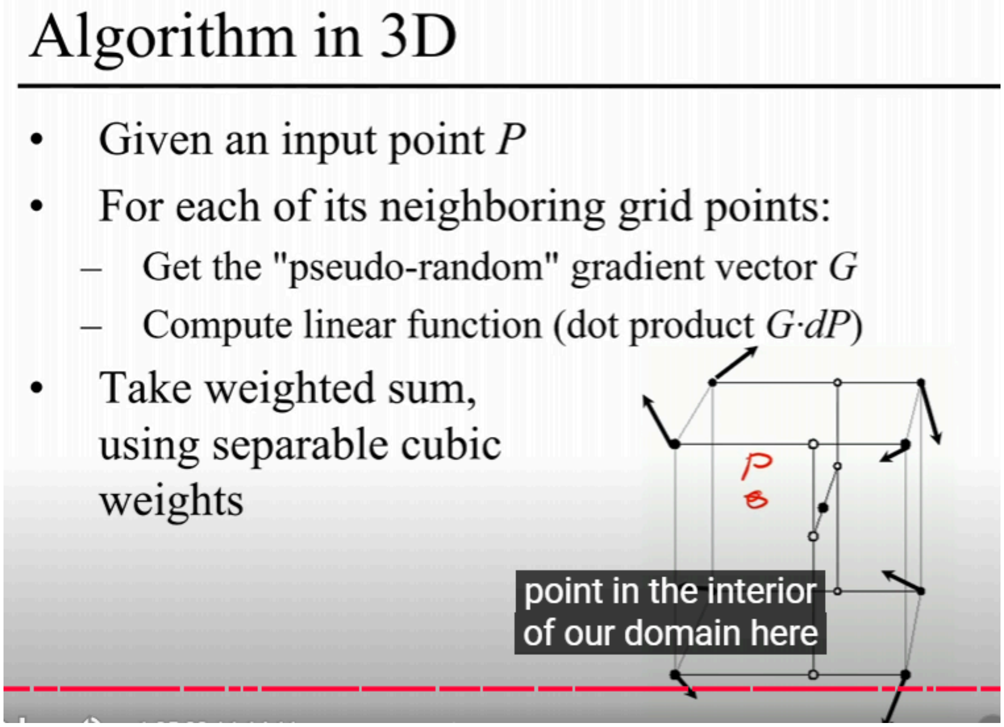
1D Noise: Reconstruct at *P*

- Compute the values from the two neighboring linear functions: $n_1 = dx \cdot G_1$; $n_2 = (dx - 1) \cdot G_2$
- Weights
  $w_1 = 3dx^2 - 2dx^3$ and $w_2 =$
  $3(1-dx)2-2(1-dx)3$

  i.e.: noise $= w_1\, G_1\, dx + w_2\, G_2\, (dx - 1)$

Just remember that you have
two zero values and two slopes,

- Perlin Noise in 3D

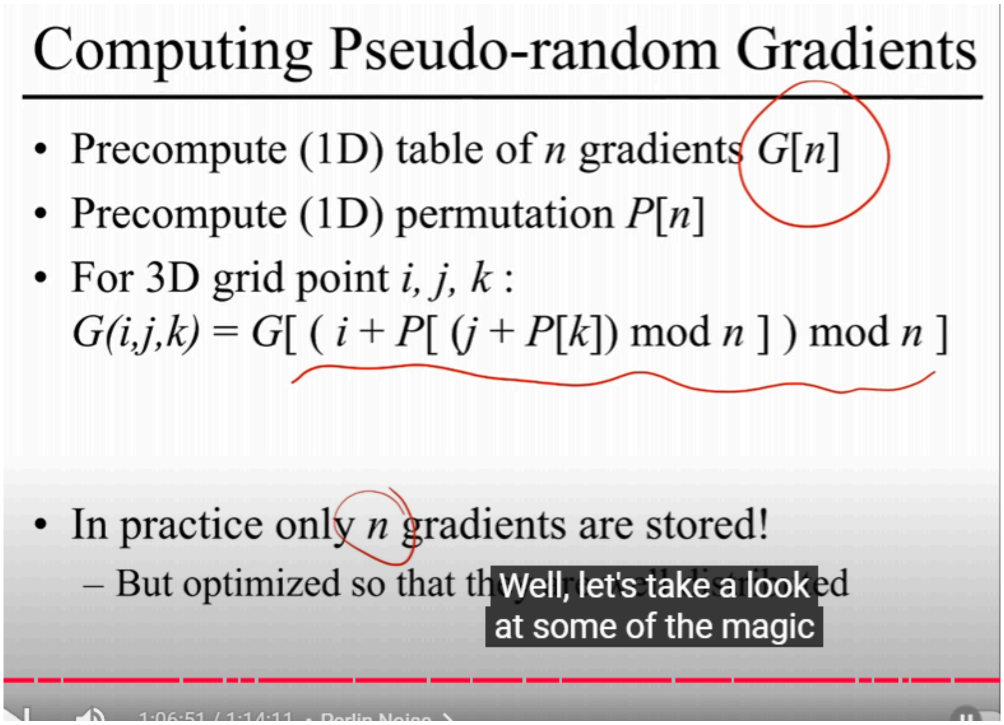


Algorithm in 3D

- Given an input point P
- For each of its neighboring grid points:
  - Get the "pseudo-random" gradient vector G
  - Compute linear function (dot product $G{\cdot}dP$)
- Take weighted sum, using separable cubic weights

point in the interior of our domain here

- Compute perlin noise



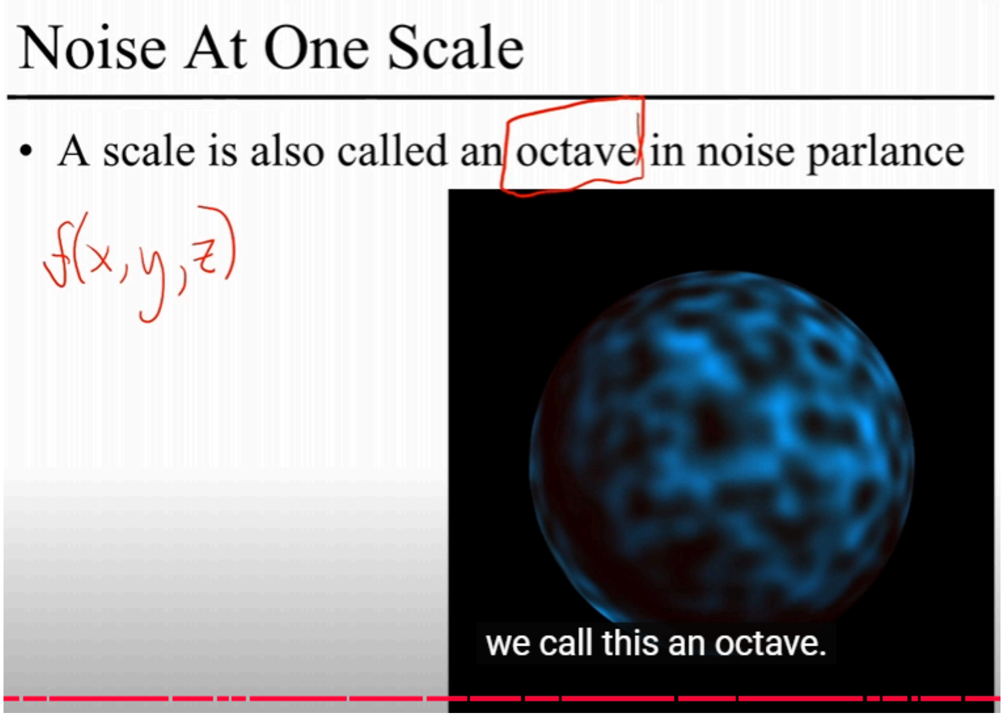
Computing Pseudo-random Gradients

- Precompute (1D) table of n gradients $G[n]$
- Precompute (1D) permutation $P[n]$
- For 3D grid point i, j, k :
  $G(i,j,k) = G[\ (\ i + P[\ (j + P[k]) \bmod n\ ]\ )\ \bmod n\ ]$

- In practice only n gradients are stored!
  - But optimized so that th ed

Well, let's take a look at some of the magic

- Example

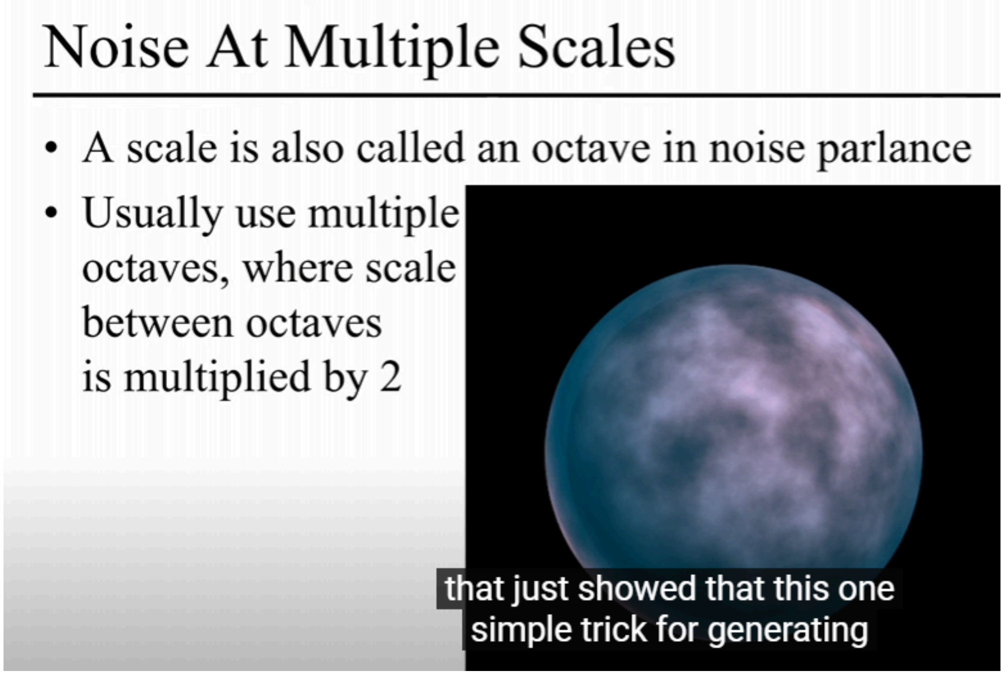


Noise At One Scale

- A scale is also called an octave in noise parlance

$f(x,y,z)$

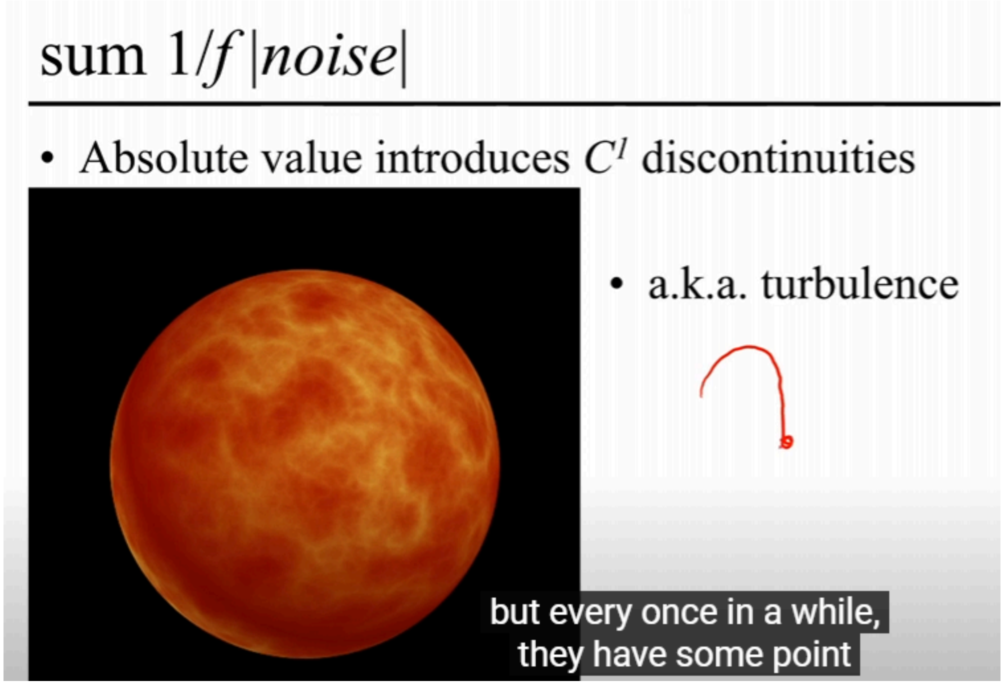
we call this an octave.



Noise At Multiple Scales

- A scale is also called an octave in noise parlance
- Usually use multiple octaves, where scale between octaves is multiplied by 2

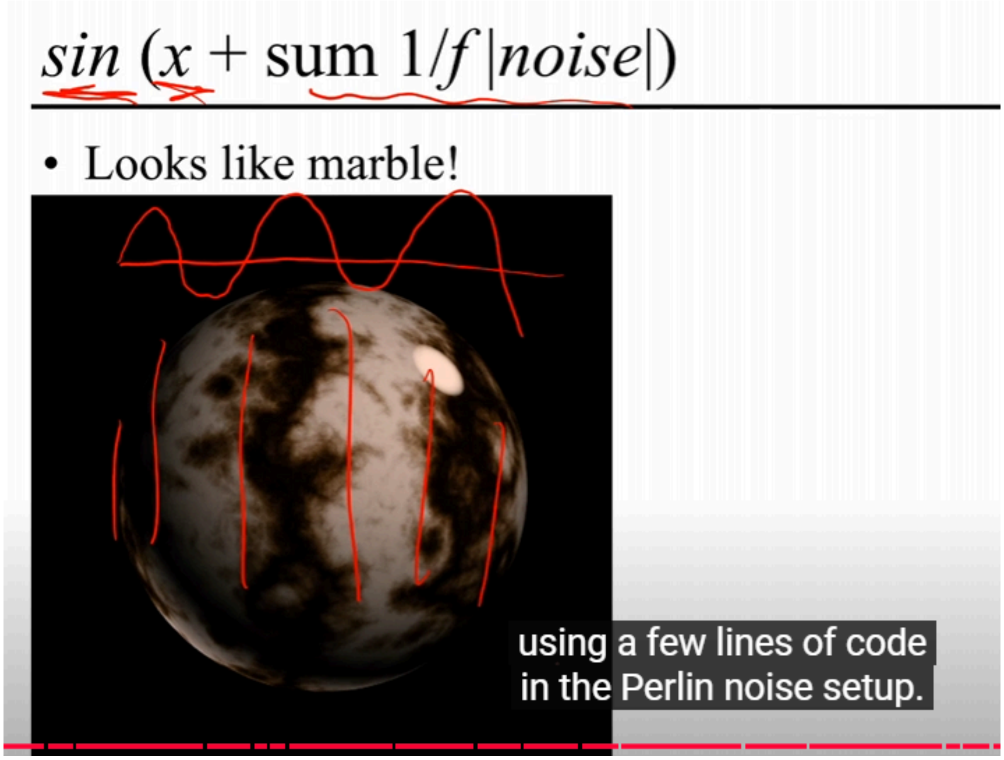
that just showed that this one
simple trick for generating



sum $1/f\,|noise|$

- Absolute value introduces C^1 discontinuities
- a.k.a. turbulence

but every once in a while,
they have some point

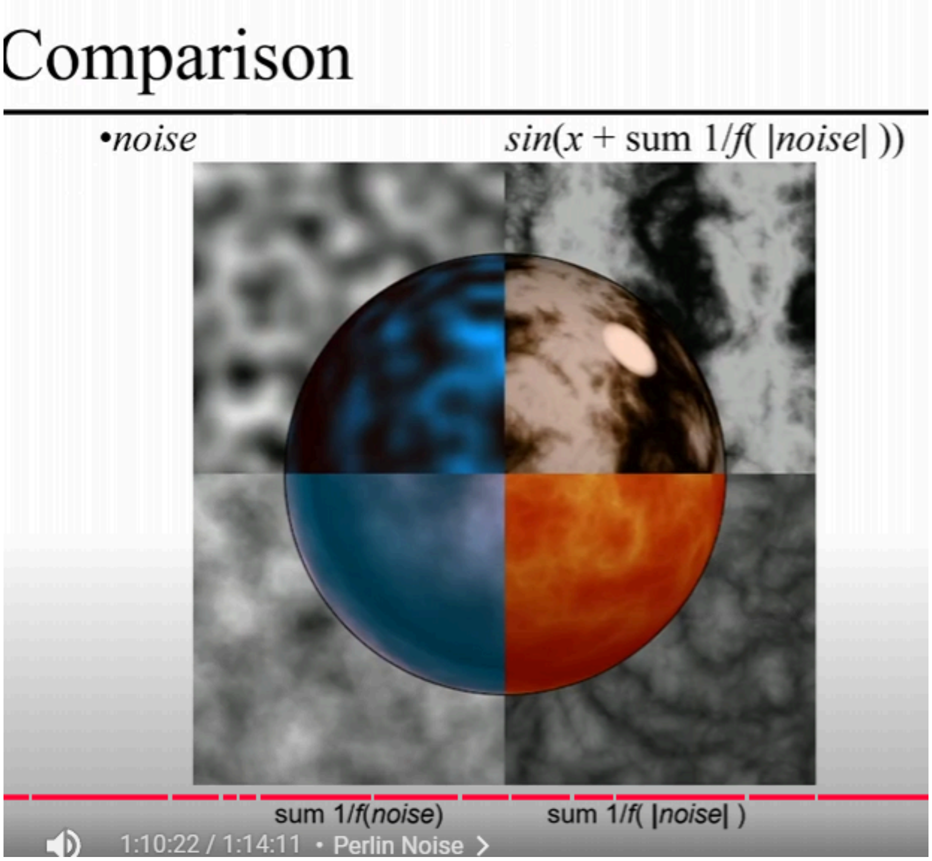


$\underline{sin}\ (\underline{x} + \underline{\text{sum } 1/f\,|noise|})$

- Looks like marble!

using a few lines of code
in the Perlin noise setup.

- Comparison

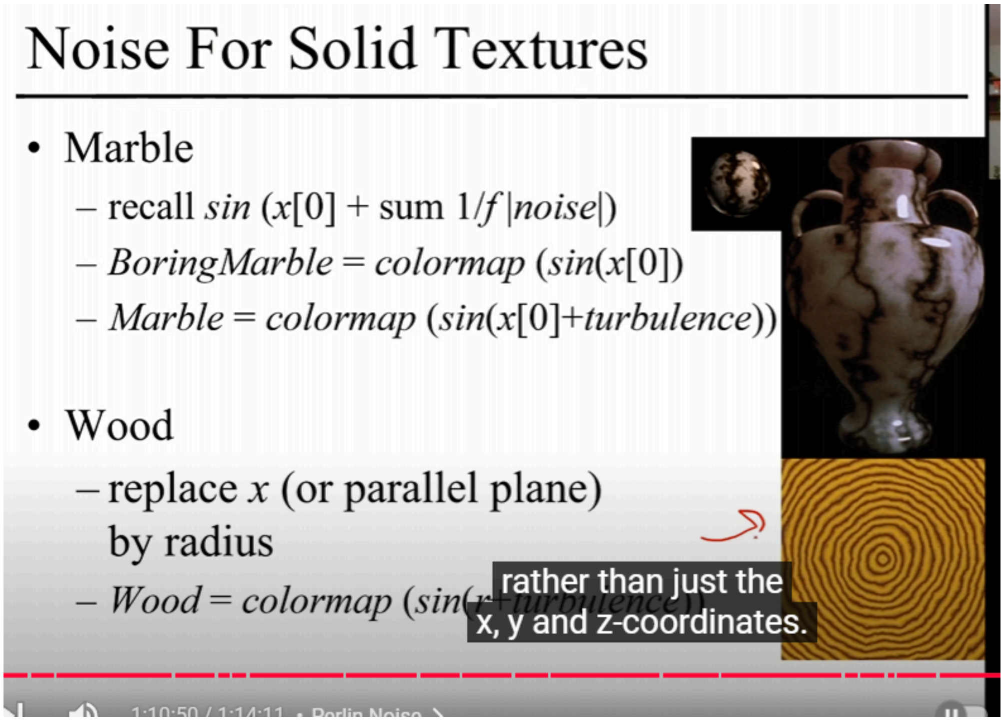


Comparison

•*noise* — *sin*(*x* + sum 1/*f*(|*noise*|))

sum 1/*f*(*noise*) — sum 1/*f*(|*noise*|)

- For solid Textures

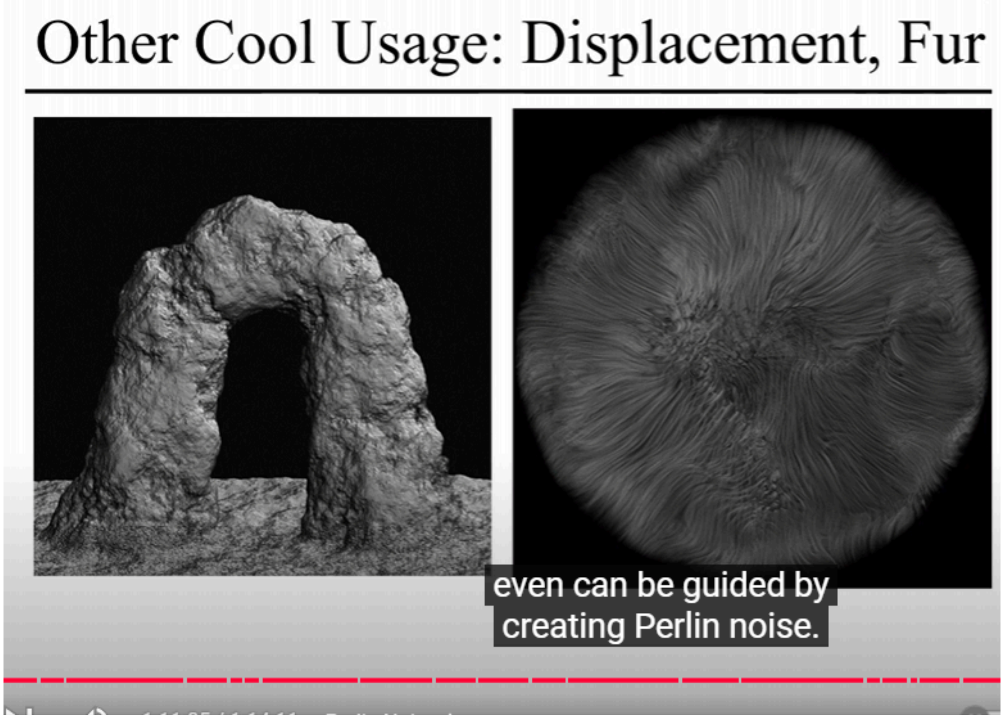


Noise For Solid Textures

- Marble
  - recall $\sin(x[0] + \text{sum } 1/f\,|noise|)$
  - $BoringMarble = colormap(\sin(x[0]))$
  - $Marble = colormap(\sin(x[0]+turbulence))$
- Wood
  - replace x (or parallel plane) by radius
  - $Wood = colormap(\sin(r+turbulence))$

rather than just the
x, y and z-coordinates.

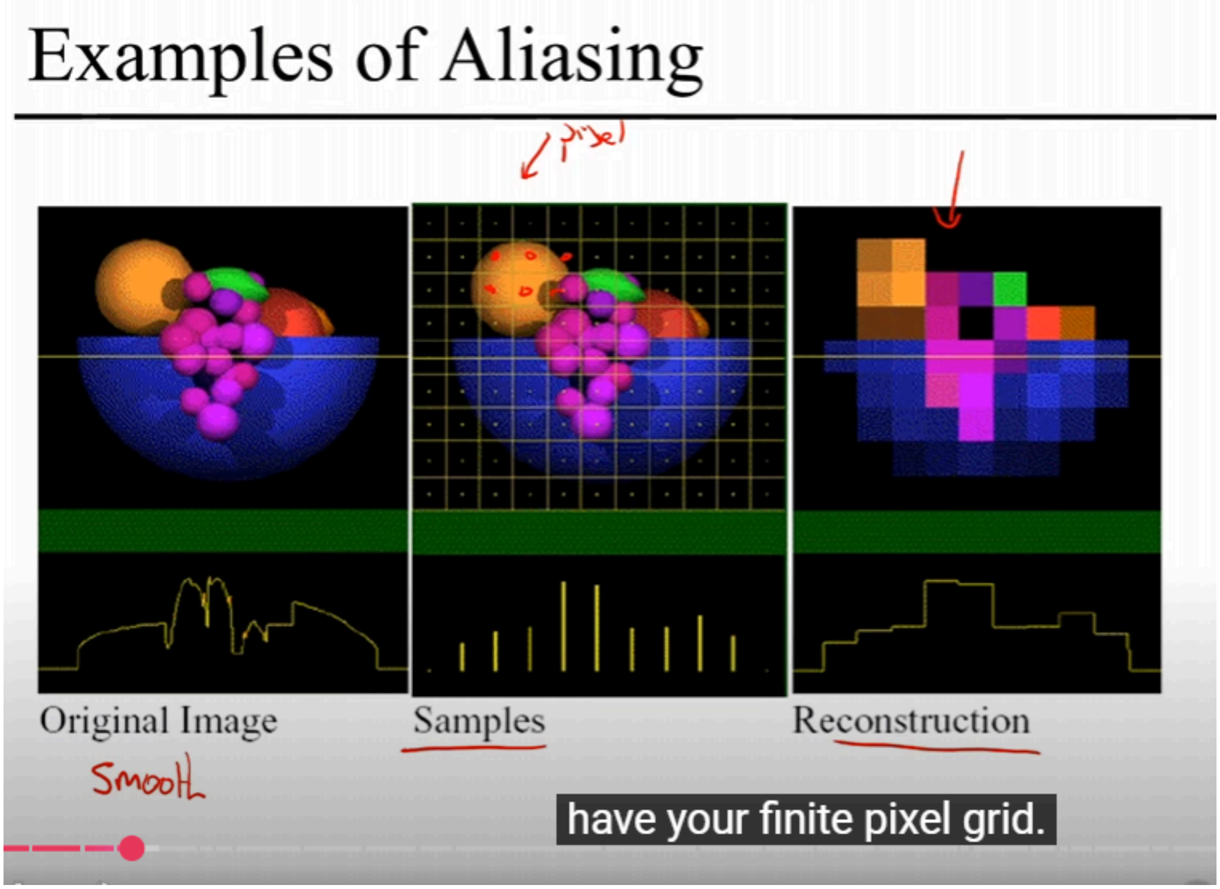
        - Fur



Other Cool Usage: Displacement, Fur

even can be guided by creating Perlin noise.

- **L15: Antialiasing; Sampling and Reconstruction**
    - Example of Aliasing

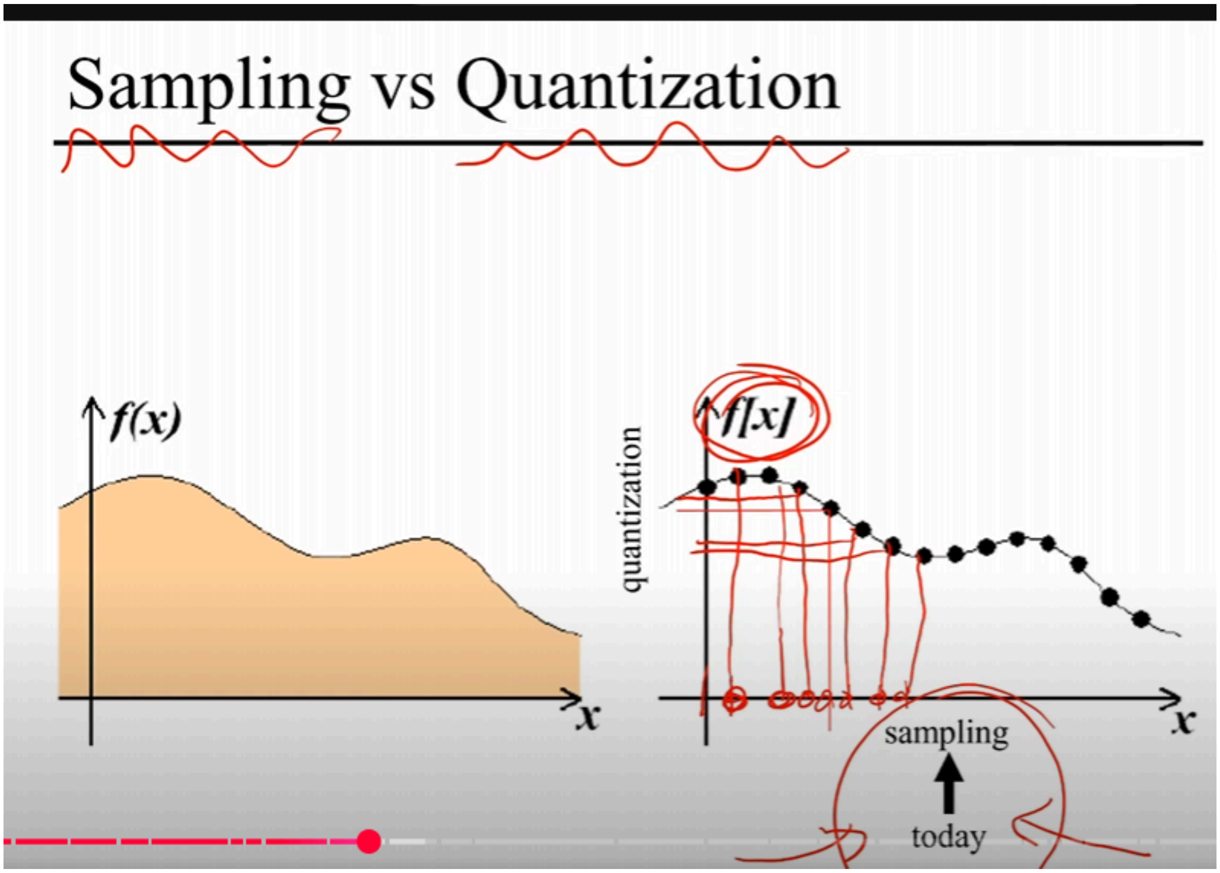


Examples of Aliasing

have your finite pixel grid.

        - Aliasing appears as jagged edges, moiré patterns, or incorrect details.

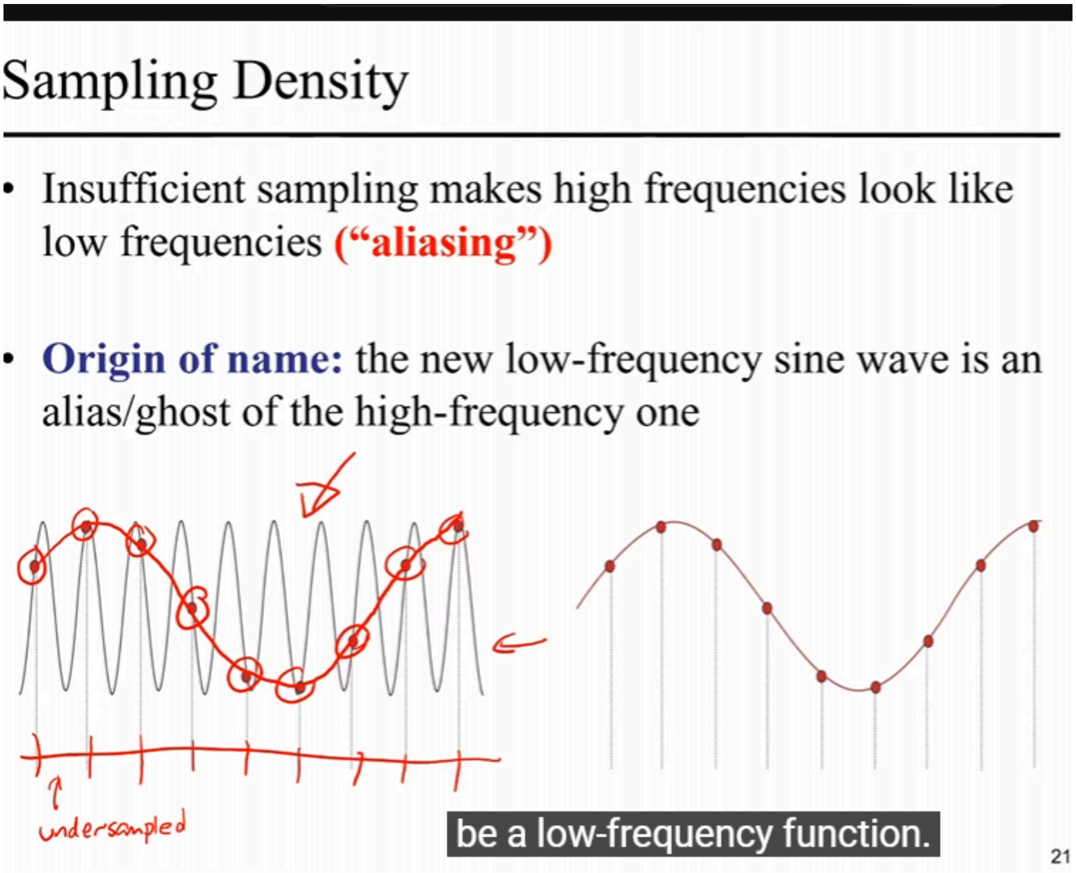
- Sampling vs Quantization



Sampling vs Quantization

  - Sampling
    - Mapping a continuous function to a discrete one

  - Sampling Density



Sampling Density

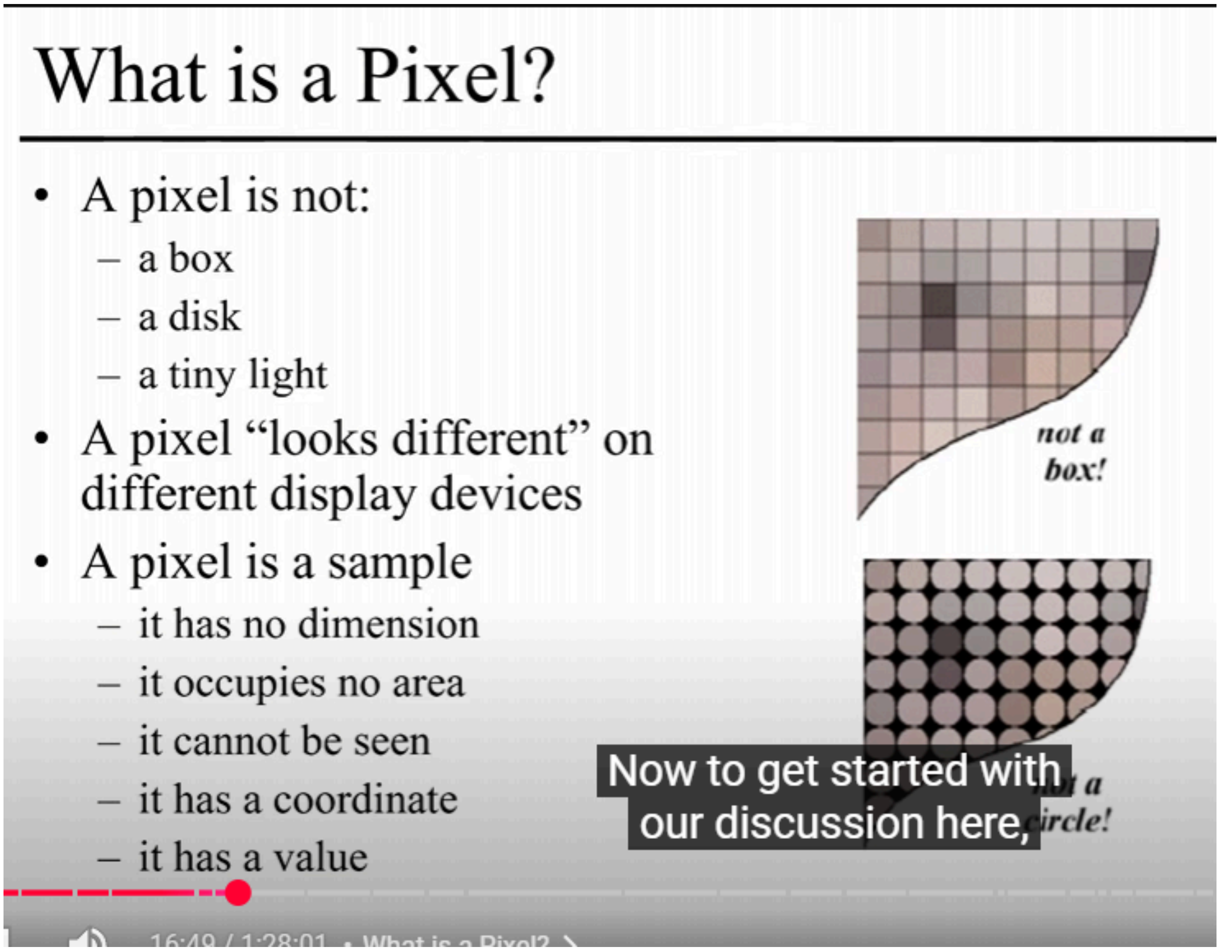
- Insufficient sampling makes high frequencies look like low frequencies **("aliasing")**
- **Origin of name:** the new low-frequency sine wave is an alias/ghost of the high-frequency one

be a low-frequency function.

21

- Quantization
  - Mapping a continuous function to a discrete one

- Pixel

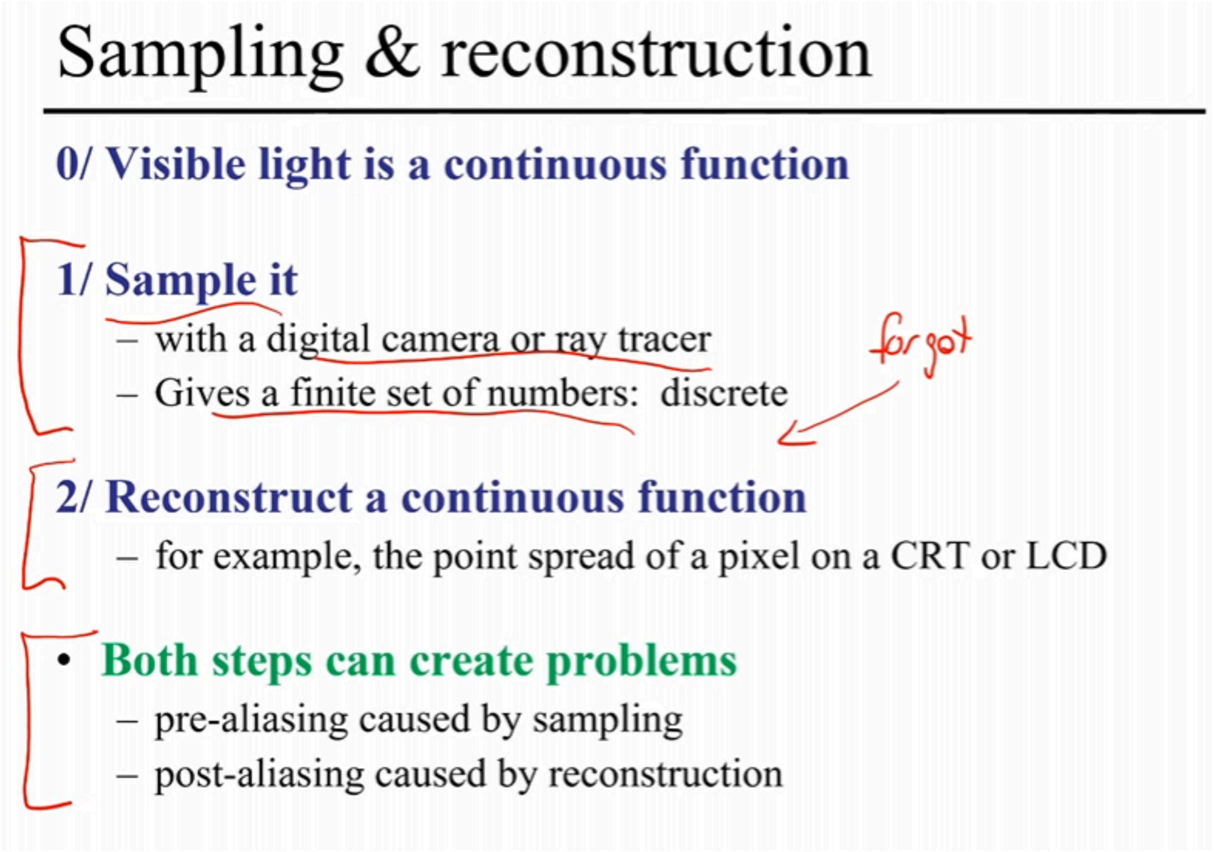


What is a Pixel?

- A pixel is not:
  - a box
  - a disk
  - a tiny light
- A pixel "looks different" on different display devices
- A pixel is a sample
  - it has no dimension
  - it occupies no area
  - it cannot be seen
  - it has a coordinate
  - it has a value

Now to get started with our discussion here,

- Reason of Aliasing



Sampling & reconstruction

0/ Visible light is a continuous function

1/ Sample it

- with a digital camera or ray tracer
- Gives a finite set of numbers: discrete

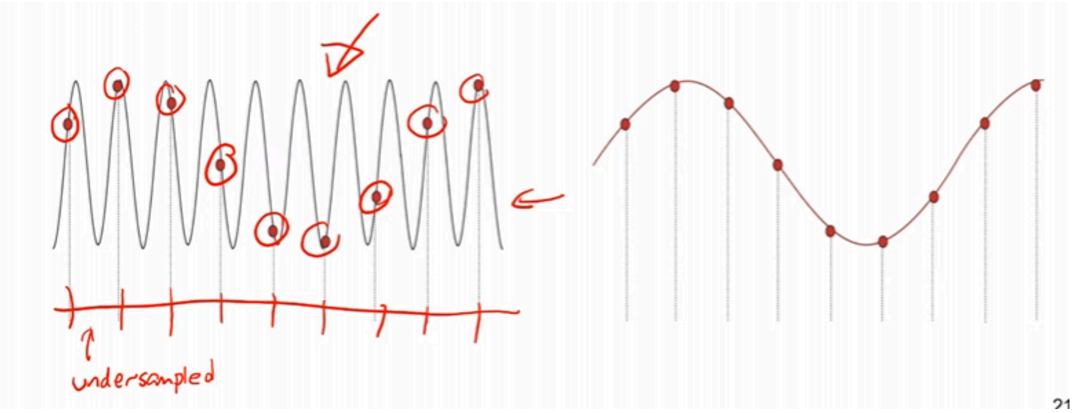
forgot

2/ Reconstruct a continuous function

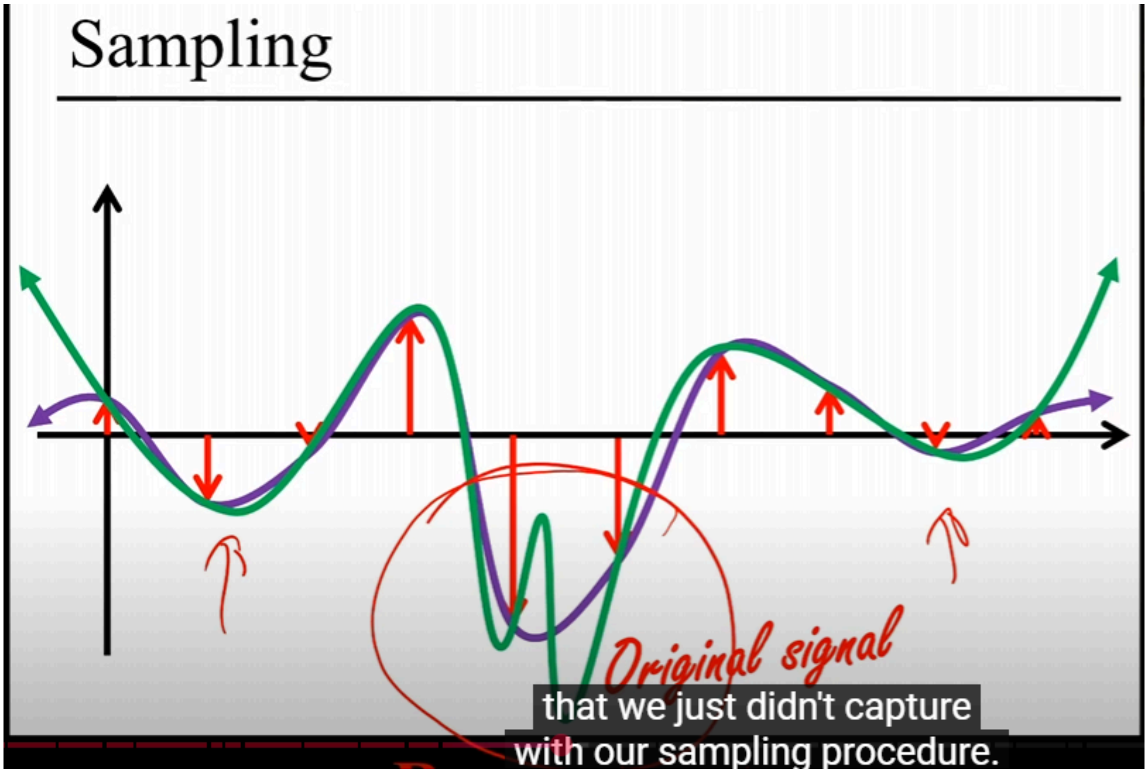
- for example, the point spread of a pixel on a CRT or LCD

- **Both steps can create problems**
  - pre-aliasing caused by sampling
  - post-aliasing caused by reconstruction

- Insufficient Sampling
  - Make high frequencies look like low frequencies)Aliasing



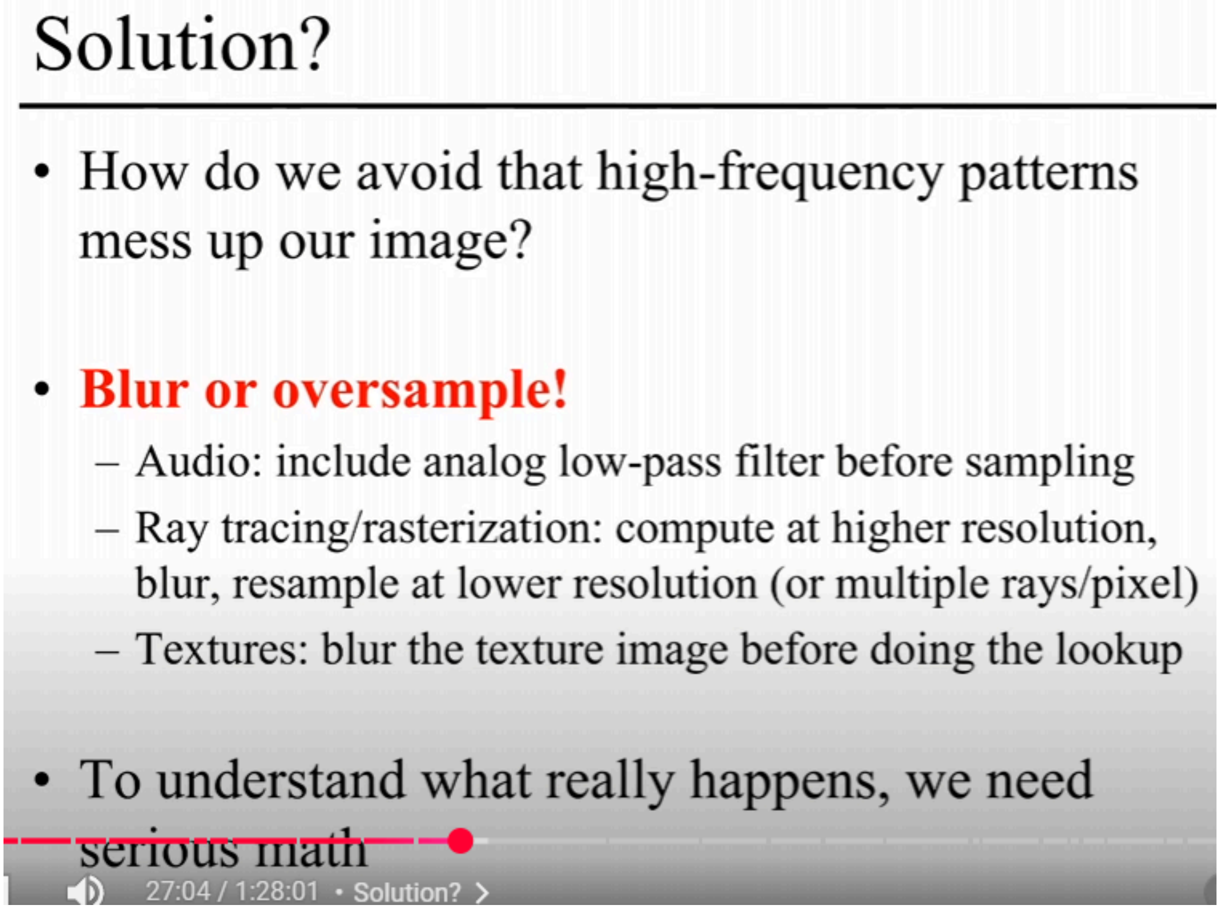
- Step 1: Sample the Function (Red Arrow)



that we just didn't capture
with our sampling procedure.

- Step 2: Reconstruct a continuous Function (Purple Line)
  - which is different from original green line (data loss)

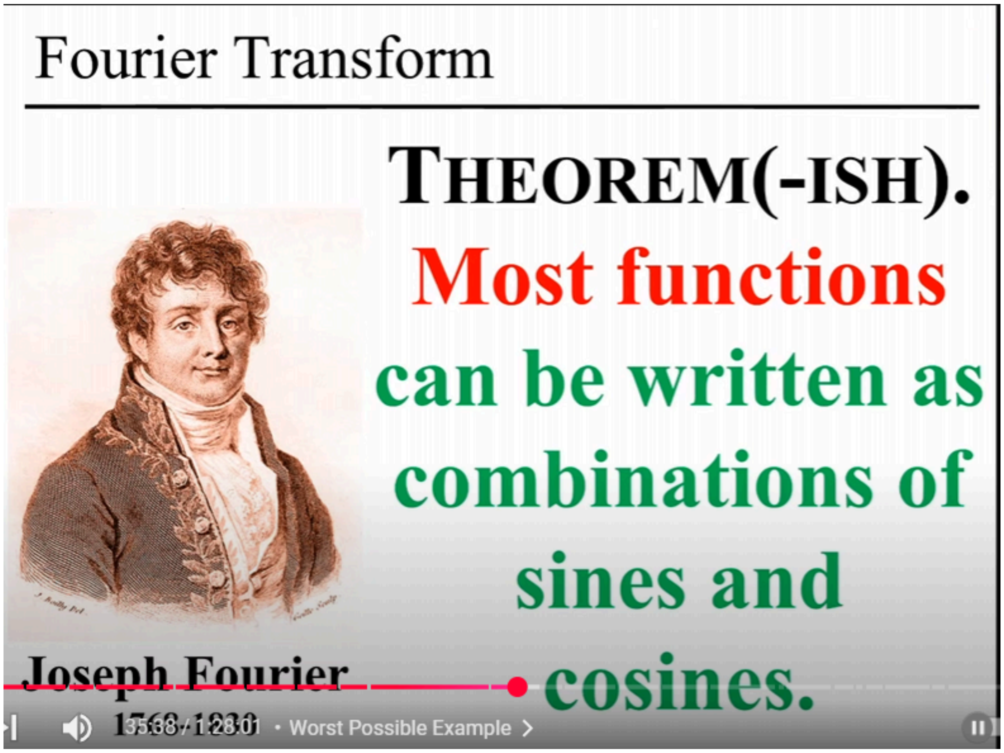
- Solution



Solution?

- How do we avoid that high-frequency patterns mess up our image?
- **Blur or oversample!**
  - Audio: include analog low-pass filter before sampling
  - Ray tracing/rasterization: compute at higher resolution, blur, resample at lower resolution (or multiple rays/pixel)
  - Textures: blur the texture image before doing the lookup
- To understand what really happens, we need serious math

  - Blue or Oversample
    - Theoretical
      - Fourier Transform: For perfect reconstruction

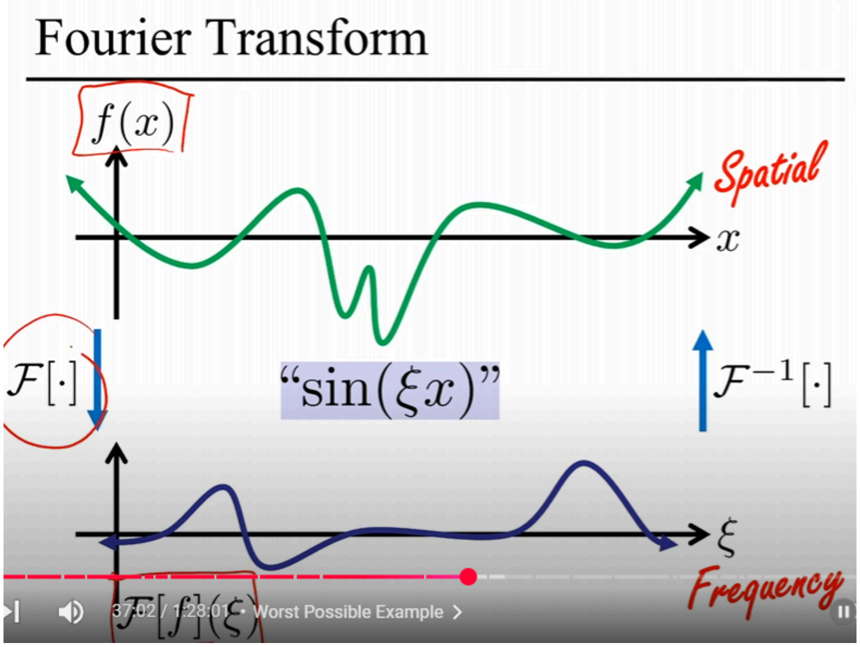


Fourier Transform

THEOREM(-ISH). **Most functions can be written as combinations of sines and cosines.**

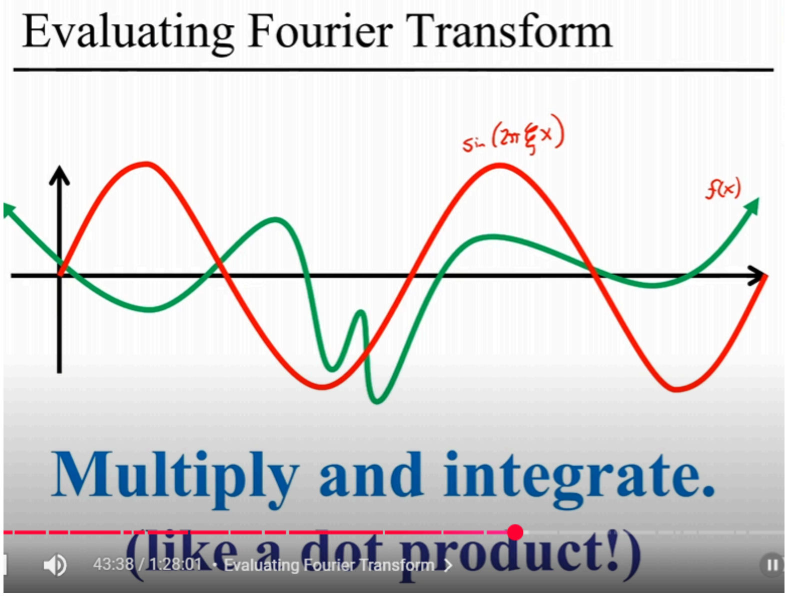
        - Any function can be combination of sin and cosina function

- Transform the Image into the Frequency Domain



Fourier Transform

  - Apply a 2D Fourier Transform (e.g., Fast Fourier Transform, FFT) to the image.
    - This decomposes the image into its frequency components, where low frequencies represent smooth variations and high frequencies represent sharp edges and details.
  - Take dot product with the Fourier and the original function

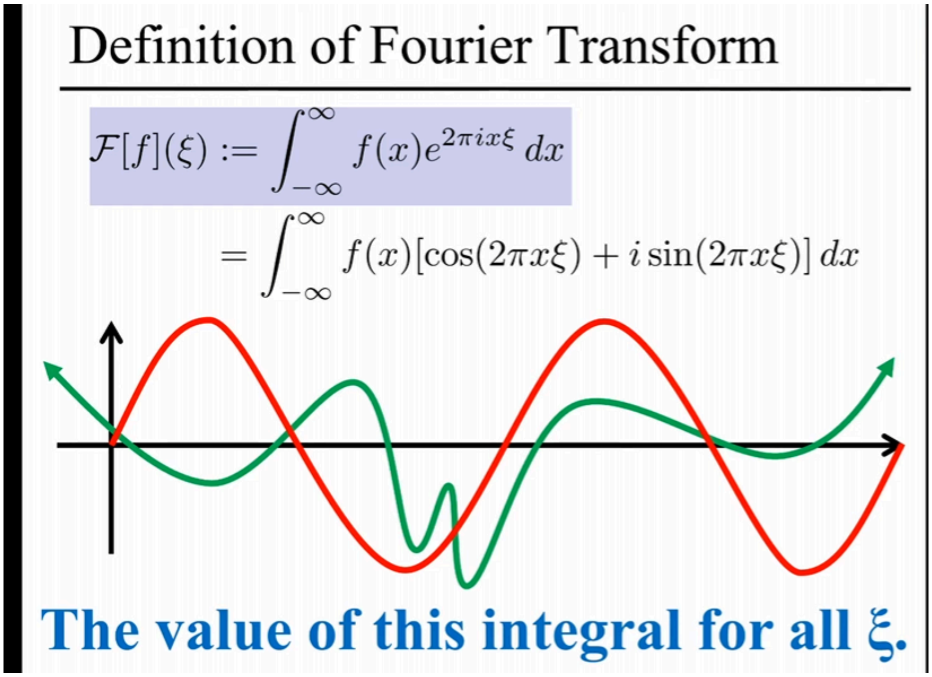


Evaluating Fourier Transform

Multiply and integrate.
(like a dot product!)

    - Tell how common (similarity) are they

- Definition of Fourier Transform

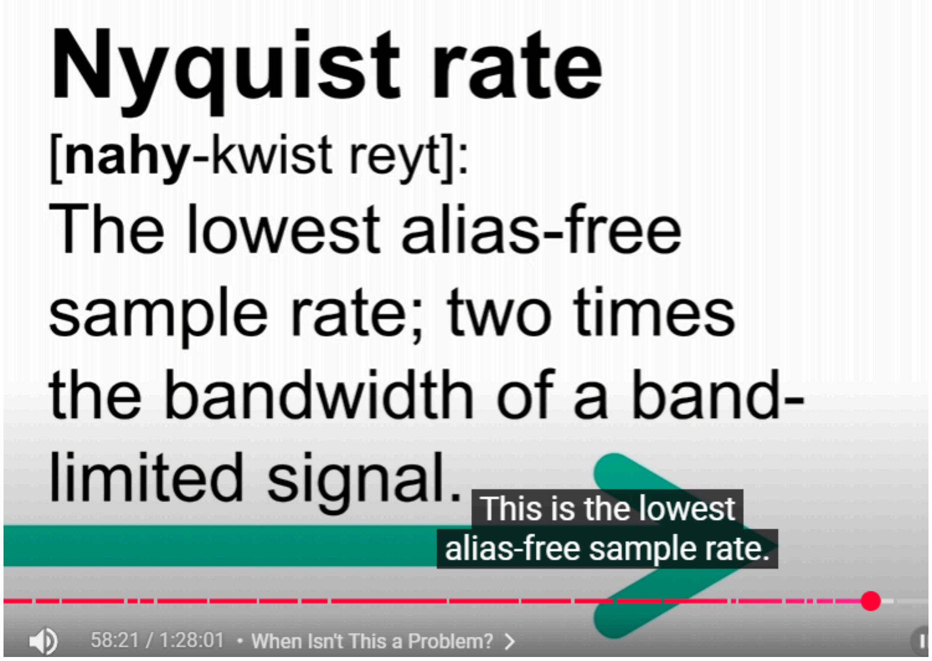


Definition of Fourier Transform

$$\mathcal{F}[f](\xi) := \int_{-\infty}^{\infty} f(x) e^{2\pi i x \xi} \, dx$$
$$= \int_{-\infty}^{\infty} f(x) [\cos(2\pi x \xi) + i \sin(2\pi x \xi)] \, dx$$

The value of this integral for all ξ.

  - How much is the frequency hiding in Orignal Function F(x)
    - By taking dot product $f(x)[\cos(2\pi x\xi)$
  - Cosine is the real part of the Fourier
  - Sine is the imaginary part
- Nyquist rate



Nyquist rate

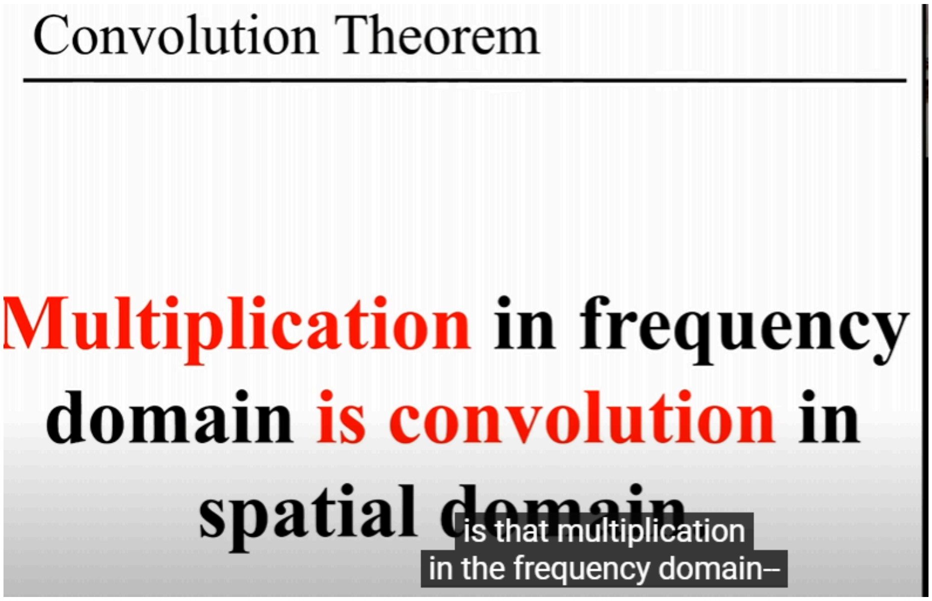
[**nahy**-kwist reyt]:

The lowest alias-free sample rate; two times the bandwidth of a band-limited signal.

This is the lowest alias-free sample rate.

58:21 / 1:28:01 • When Isn't This a Problem? >

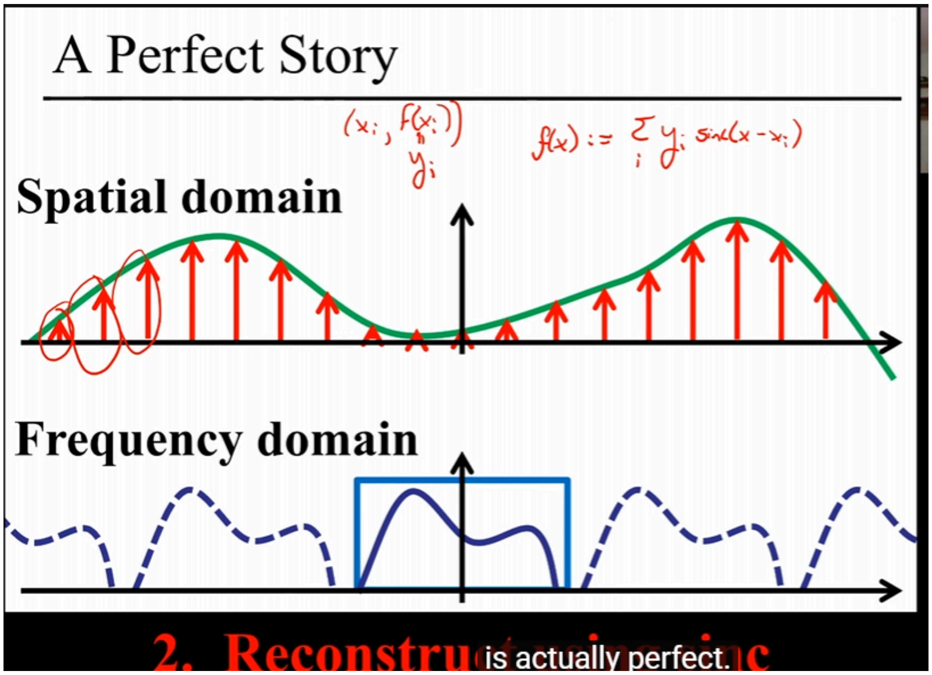
- Convolution Theorem



Convolution Theorem

Multiplication in frequency domain is convolution in spatial domain

is that multiplication
in the frequency domain–



A Perfect Story

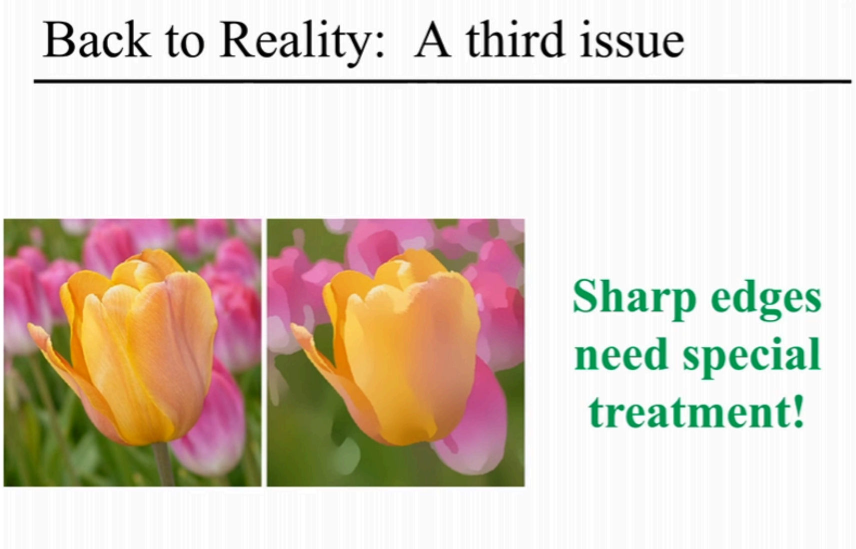
$(x_i, f(x_i))$
y_i

$f(x) := \sum_i y_i \operatorname{sinc}(x - x_i)$

2. Reconstru is actually perfect. **c**

- Not pratical
    - because practical signals cannot have finite bandwidth.
    - Neagtive lobe
    - Ifinite extent

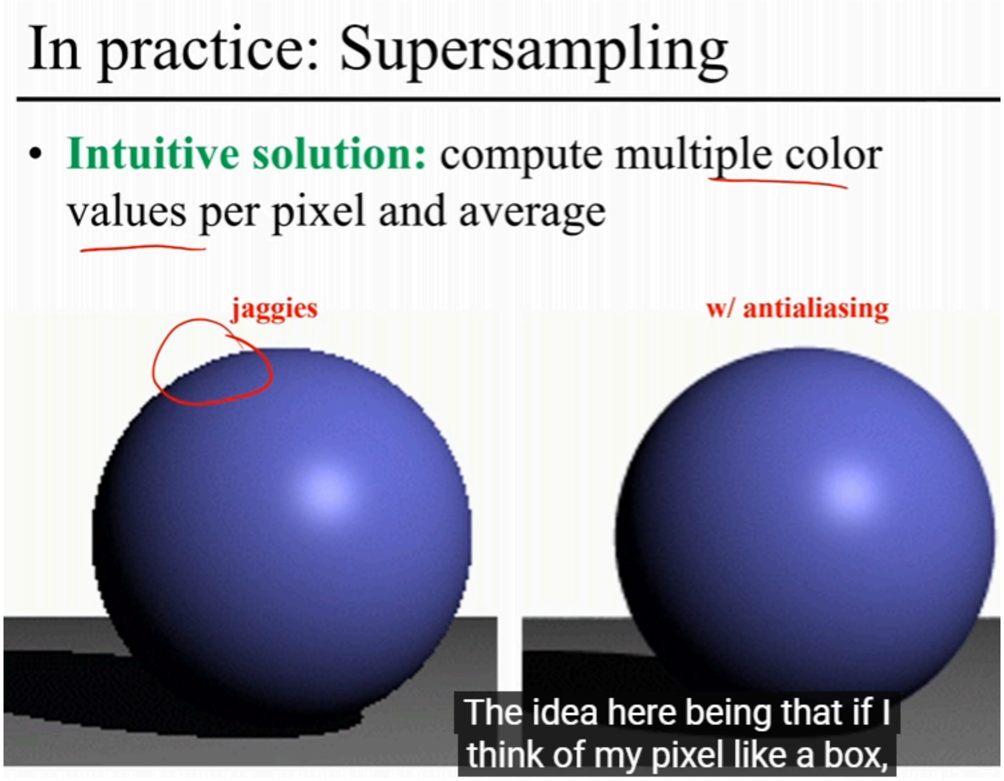
    - Sharp edges miss (Miss of High Frequency)



Back to Reality: A third issue

Sharp edges need special treatment!

- In Practice
  - Supersampling Anti-Aliasing (SSAA)



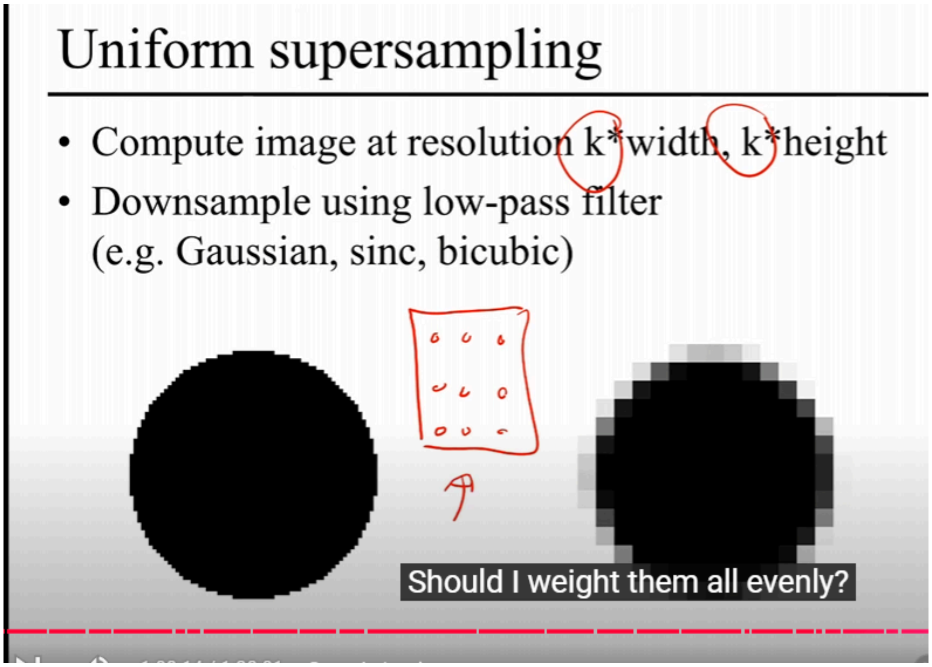
In practice: Supersampling

- **Intuitive solution:** compute multiple color values per pixel and average

The idea here being that if I think of my pixel like a box,

    - average the color in the pixel

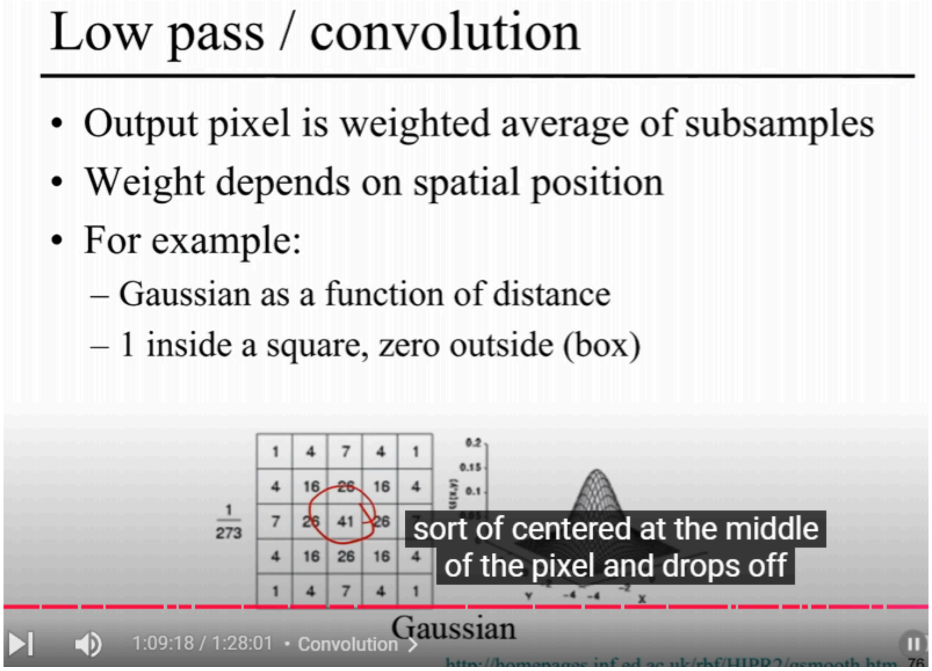
- Uniform supersampling



Uniform supersampling

- Compute image at resolution k*width, k*height
- Downsample using low-pass filter (e.g. Gaussian, sinc, bicubic)

Should I weight them all evenly?



Low pass / convolution

- Output pixel is weighted average of subsamples
- Weight depends on spatial position
- For example:
  - Gaussian as a function of distance
  - 1 inside a square, zero outside (box)

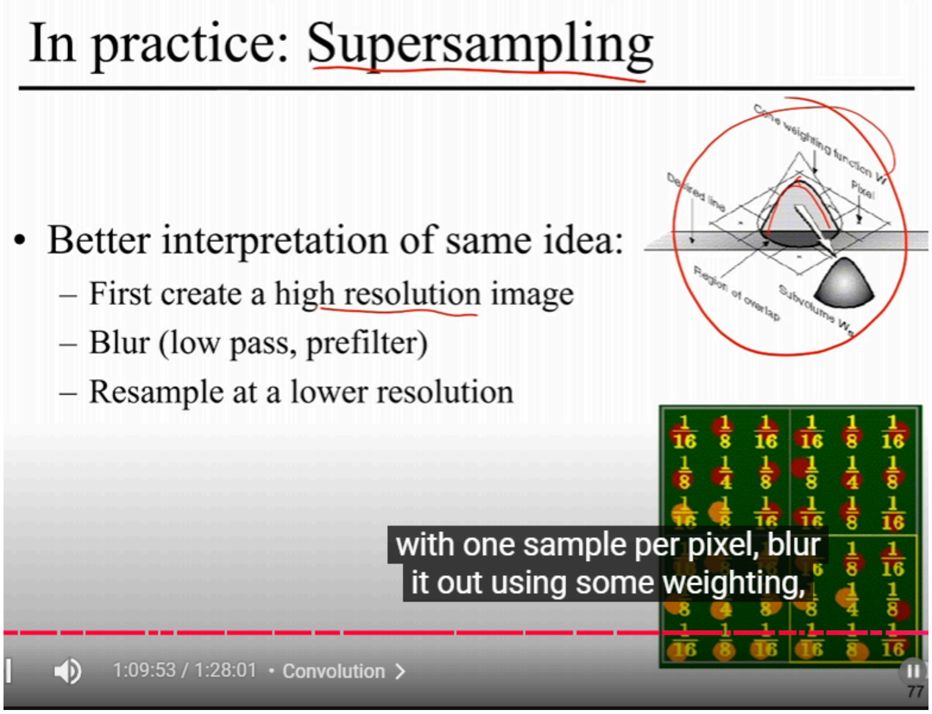
$\frac{1}{273}$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Gaussian

http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm 76

sort of centered at the middle of the pixel and drops off



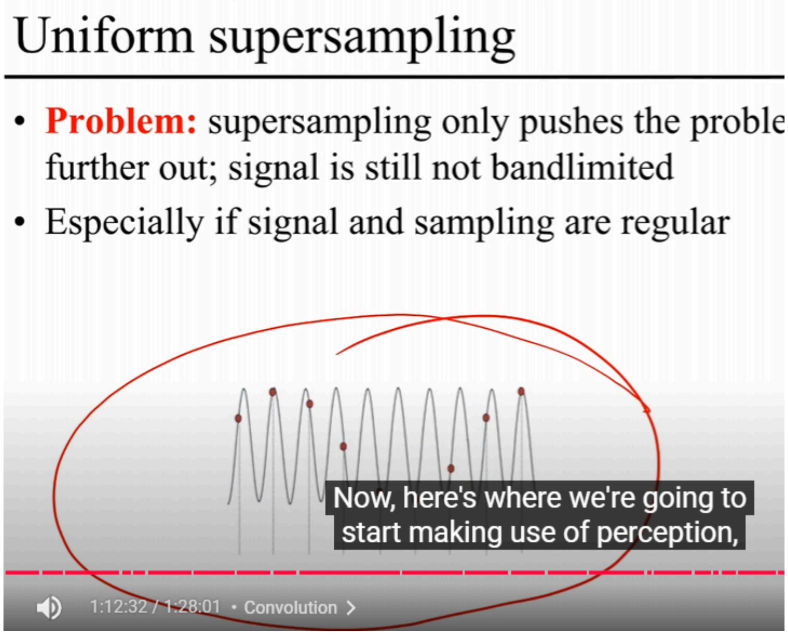
In practice: Supersampling

- Better interpretation of same idea:
  - First create a high resolution image
  - Blur (low pass, prefilter)
  - Resample at a lower resolution

with one sample per pixel, blur it out using some weighting,

- Recommended filter
  - Bicubic (piecwise polynomial): Sinc approximation
- Advantages:
  - Capture hight frequencies
  - Downsampling can use a good filter
  - Works well for edges
- Issues:
  - Frequencies above supersampling limit still aliased

    - Not good for repetitive textures

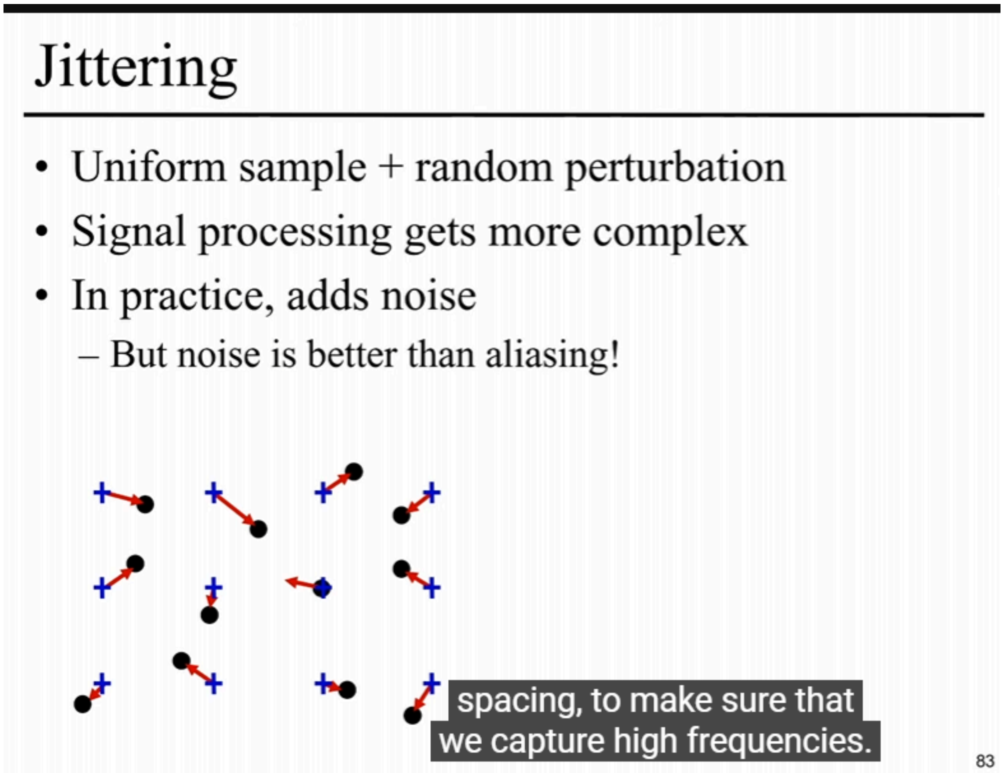


Uniform supersampling

- **Problem:** supersampling only pushes the proble further out; signal is still not bandlimited
- Especially if signal and sampling are regular

Now, here's where we're going to start making use of perception,

- Jittering

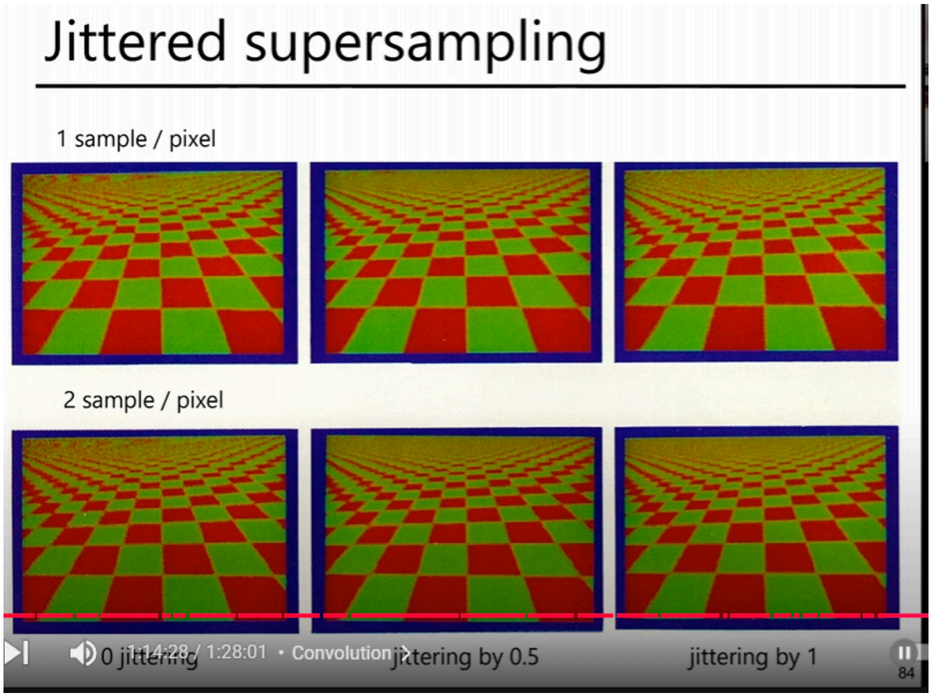


Jittering

- Uniform sample + random perturbation
- Signal processing gets more complex
- In practice, adds noise
  - But noise is better than aliasing!

spacing, to make sure that we capture high frequencies.

83



Jittered supersampling

1 sample / pixel

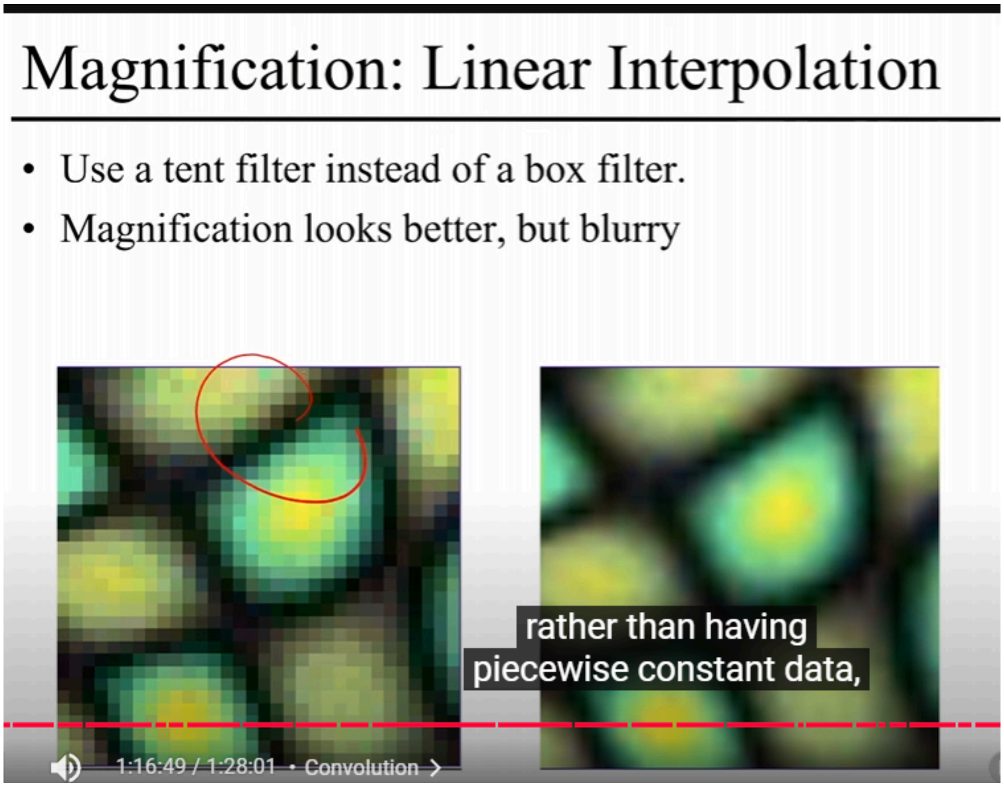
2 sample / pixel

0 jittering

jittering by 0.5

jittering by 1

- Magnification: Linear Interpolation

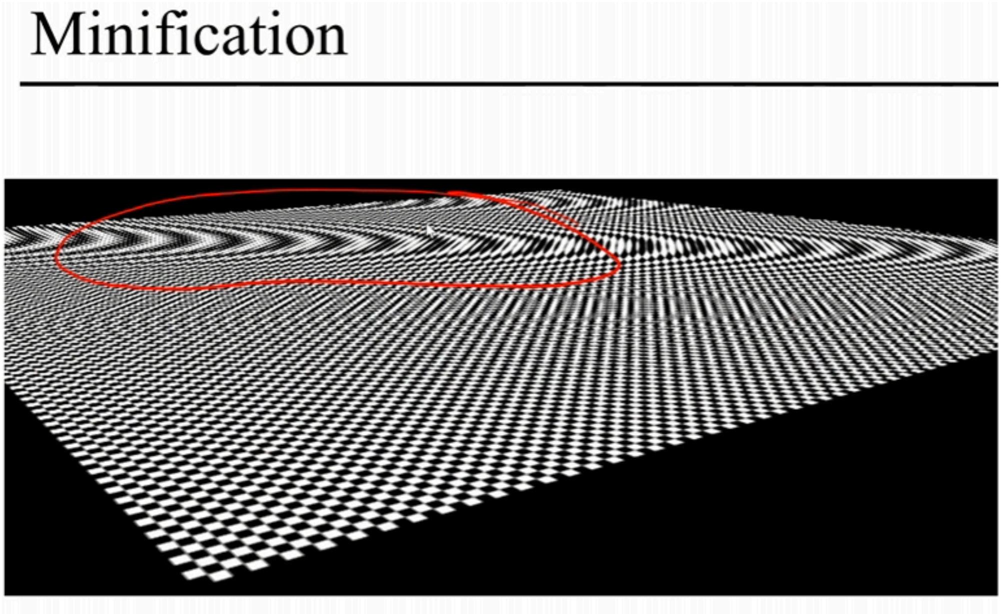


Magnification: Linear Interpolation

- Use a tent filter instead of a box filter.
- Magnification looks better, but blurry

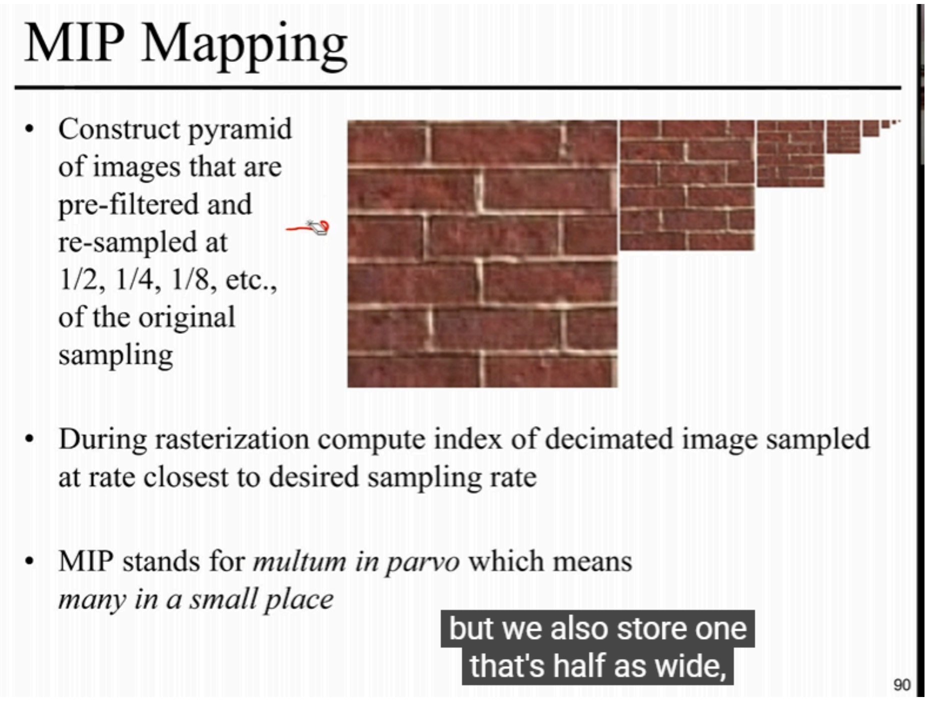
rather than having
piecewise constant data,

- Minification



Minification

- MIP Mapping



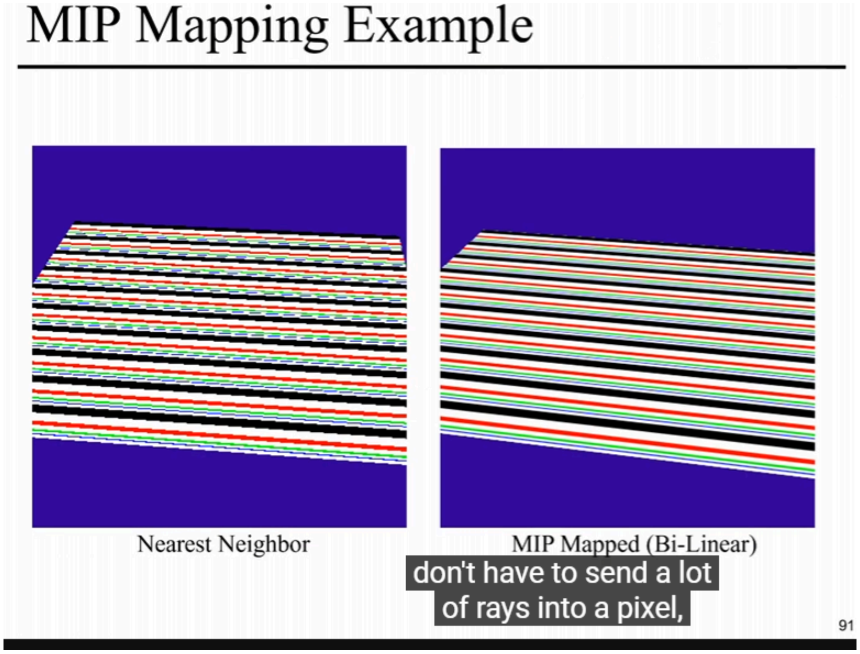
MIP Mapping

- Construct pyramid of images that are pre-filtered and re-sampled at 1/2, 1/4, 1/8, etc., of the original sampling
- During rasterization compute index of decimated image sampled at rate closest to desired sampling rate
- MIP stands for *multum in parvo* which means *many in a small place*

but we also store one that's half as wide,

90

- Example



MIP Mapping Example

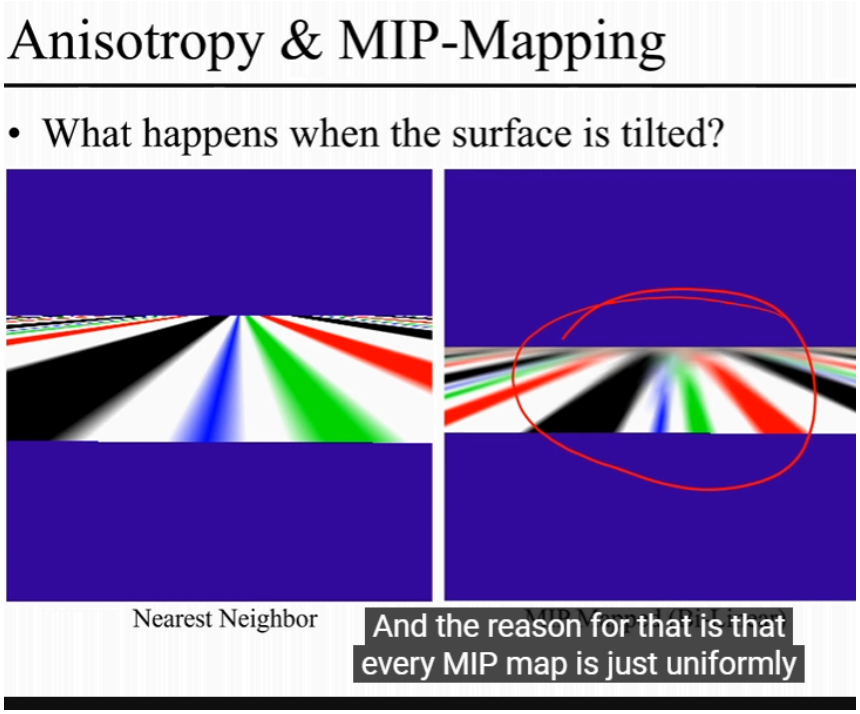
Nearest Neighbor

MIP Mapped (Bi-Linear)

don't have to send a lot of rays into a pixel,

91

- Drawback



Anisotropy & MIP-Mapping

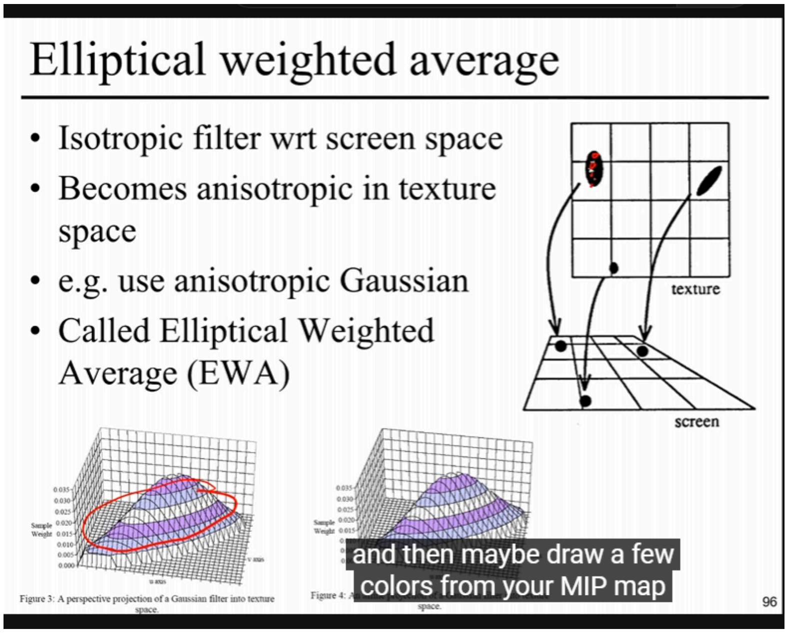
- What happens when the surface is tilted?

Nearest Neighbor

MIP Mapped (Bi-Linear)

And the reason for that is that every MIP map is just uniformly

- Fix with Elliptical Weighted Average



Elliptical weighted average

- Isotropic filter wrt screen space
- Becomes anisotropic in texture space
- e.g. use anisotropic Gaussian
- Called Elliptical Weighted Average (EWA)

Figure 3: A perspective projection of a Gaussian filter into texture space.

Figure 4: A

space.

and then maybe draw a few colors from your MIP map

96

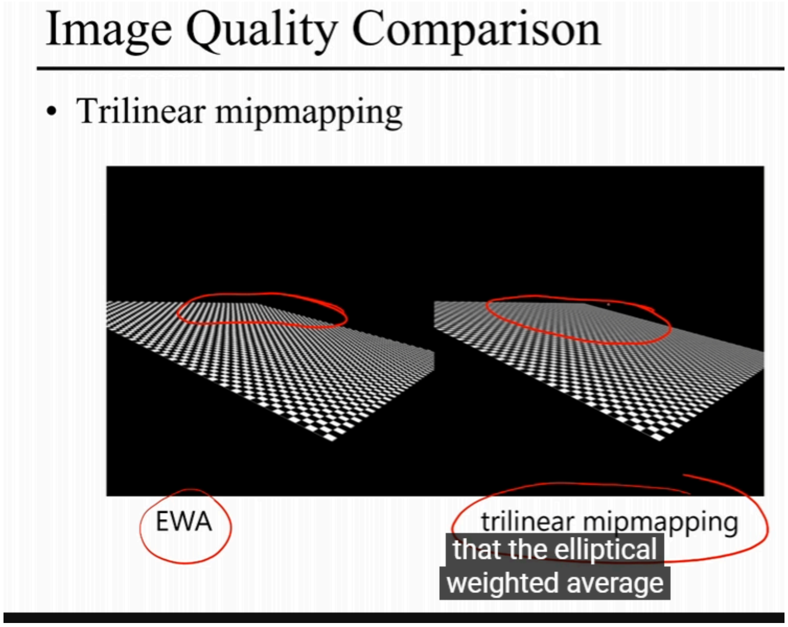
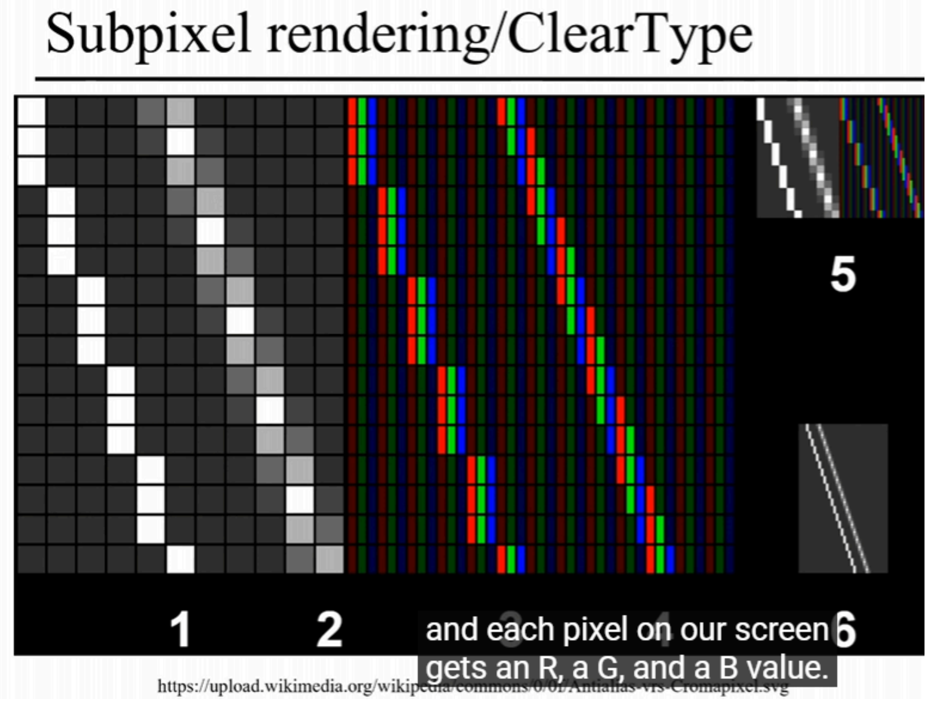


Image Quality Comparison

- Trilinear mipmapping

that the elliptical weighted average

- Subpixel rendering /ClearType for Text



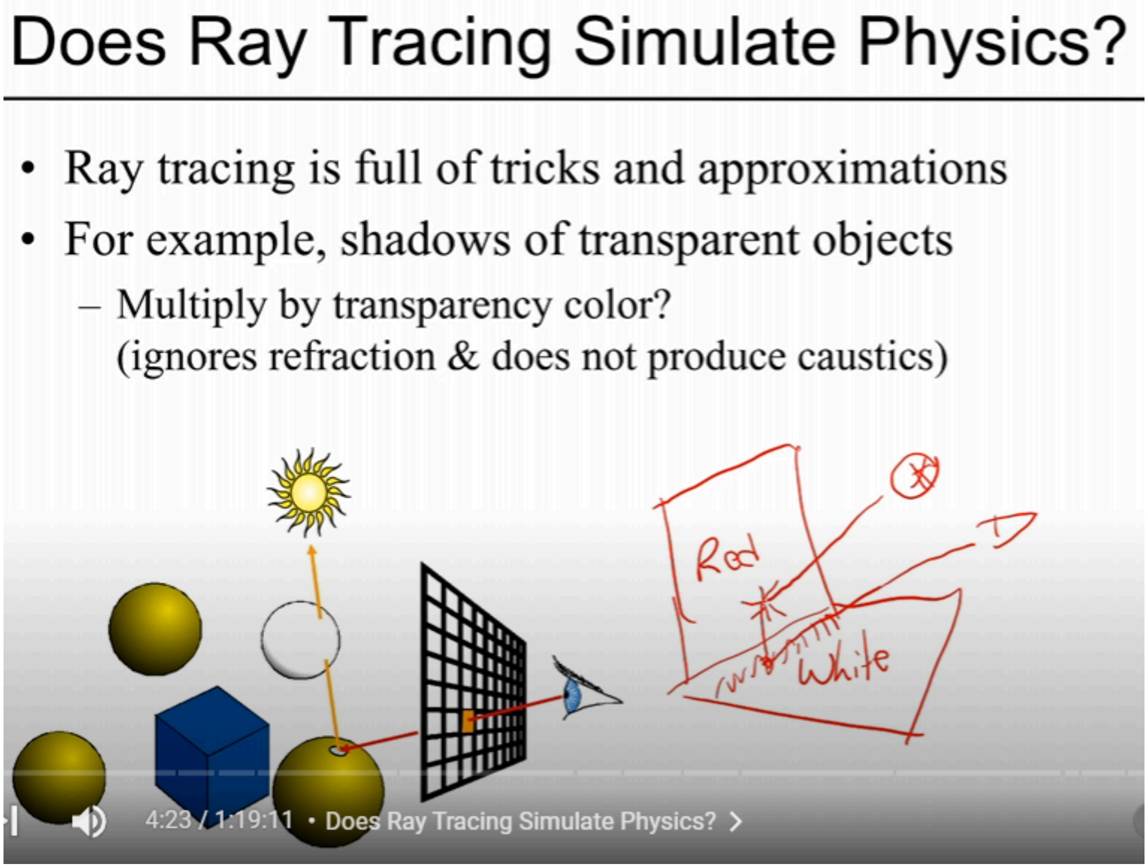
Subpixel rendering/ClearType

and each pixel on our screen gets an R, a G, and a B value.

https://upload.wikimedia.org/wikipedia/commons/0/0f/Antialias-vrs-Cromapixel.svg

    - Control the subpixel (RGB)

- **L16: Global Illumination and Monte Carlo**
  - Reason of GI
    - Does Ray Tracing Simulate Physics?



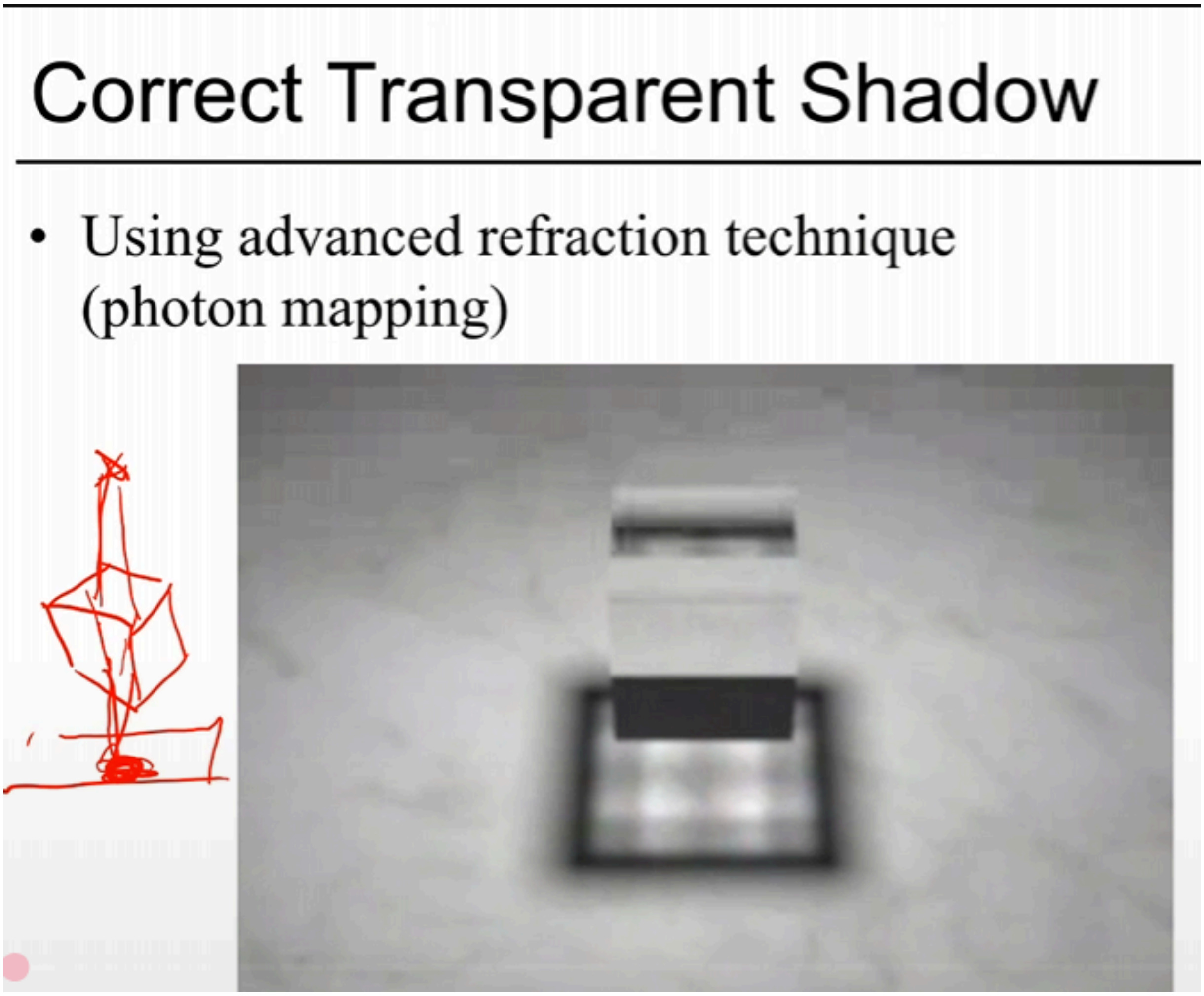
Does Ray Tracing Simulate Physics?

- Ray tracing is full of tricks and approximations
- For example, shadows of transparent objects
  - Multiply by transparency color? (ignores refraction & does not produce caustics)

4:23 / 1:19:11 • Does Ray Tracing Simulate Physics? >

    - No, It is backward ray tracing

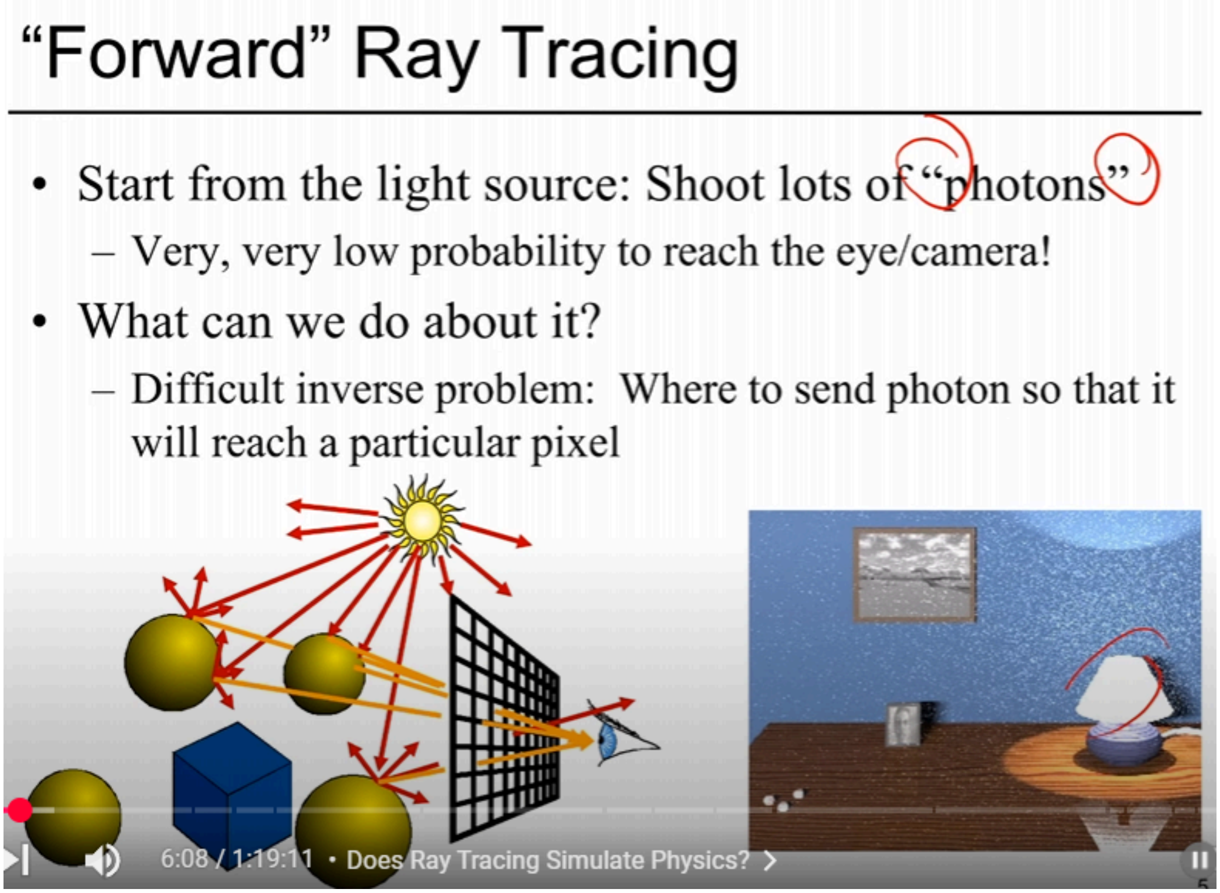
  - lot of physical
- Correct Transparent Shadow



Correct Transparent Shadow

- Using advanced refraction technique (photon mapping)

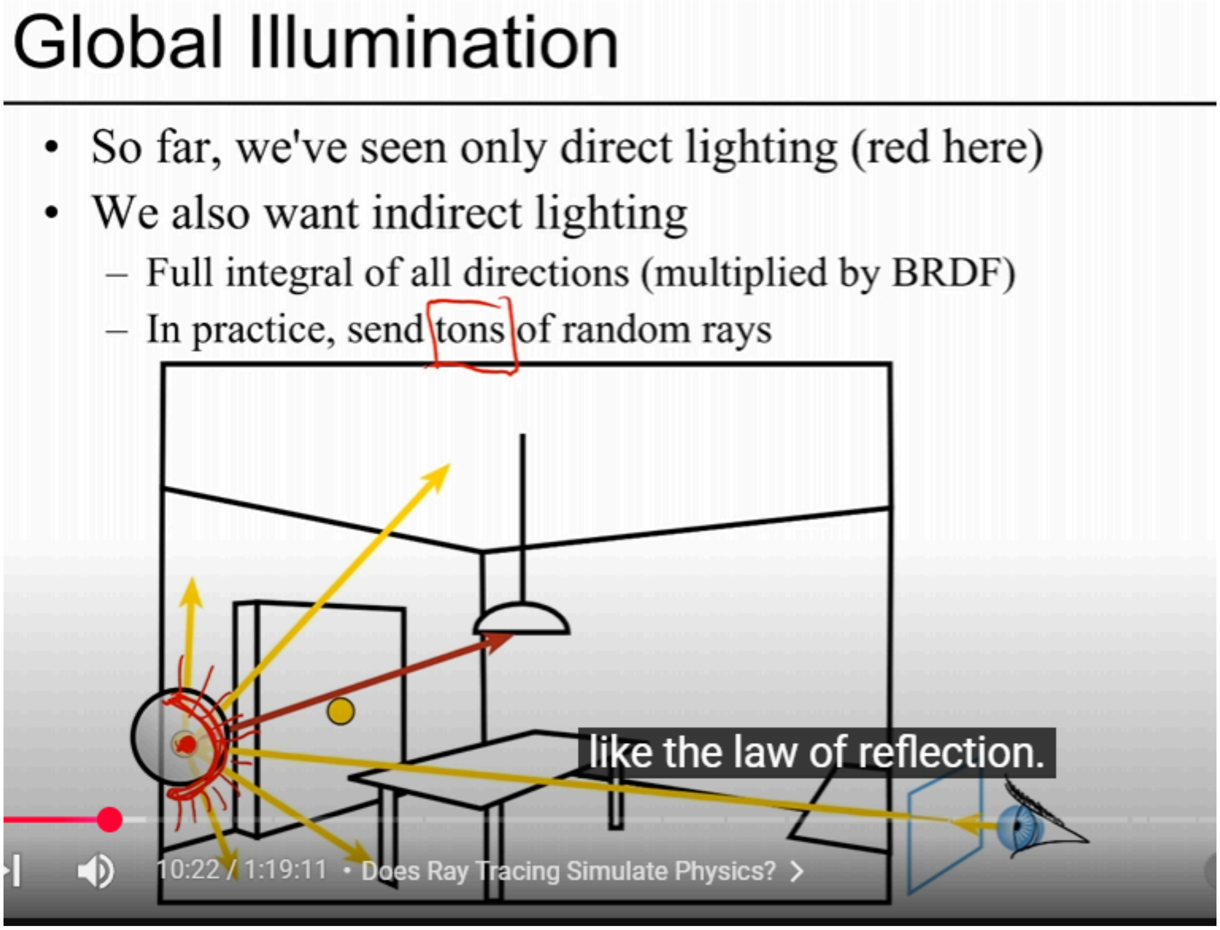
- Forward Ray Tracing



“Forward” Ray Tracing

- Start from the light source: Shoot lots of “photons”
  - Very, very low probability to reach the eye/camera!
- What can we do about it?
  - Difficult inverse problem: Where to send photon so that it will reach a particular pixel

- Global Illumination

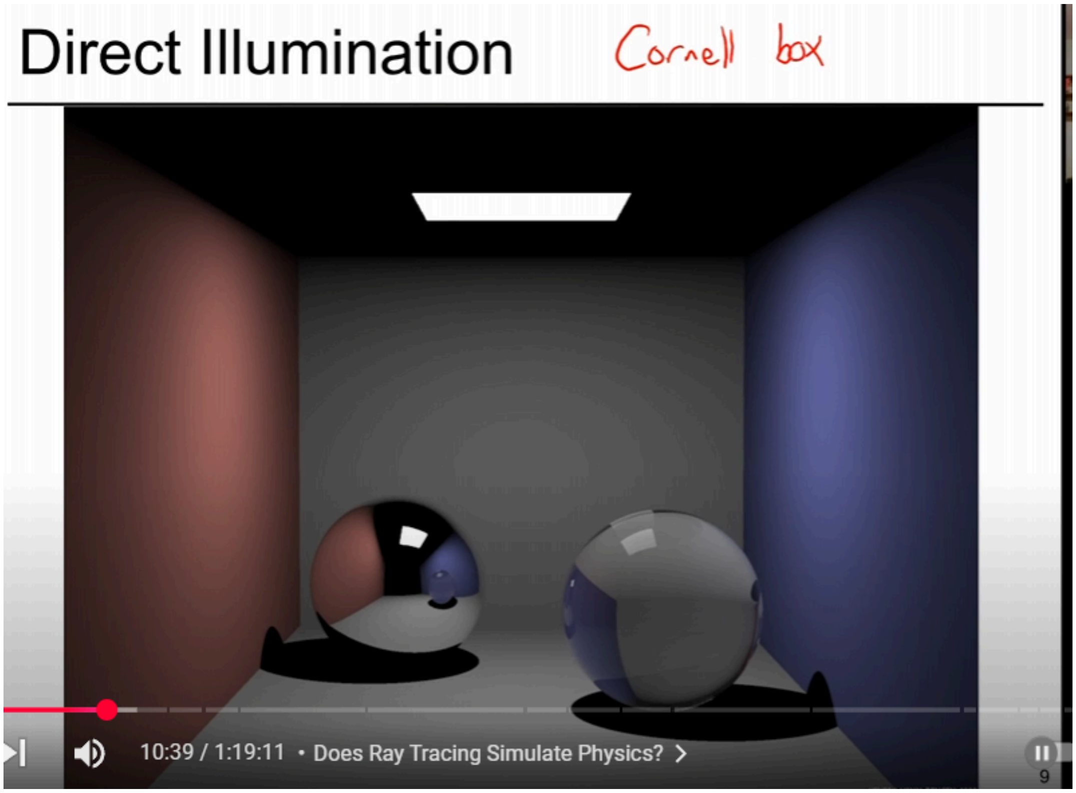


Global Illumination

- So far, we've seen only direct lighting (red here)
- We also want indirect lighting
  - Full integral of all directions (multiplied by BRDF)
  - In practice, send tons of random rays

like the law of reflection.

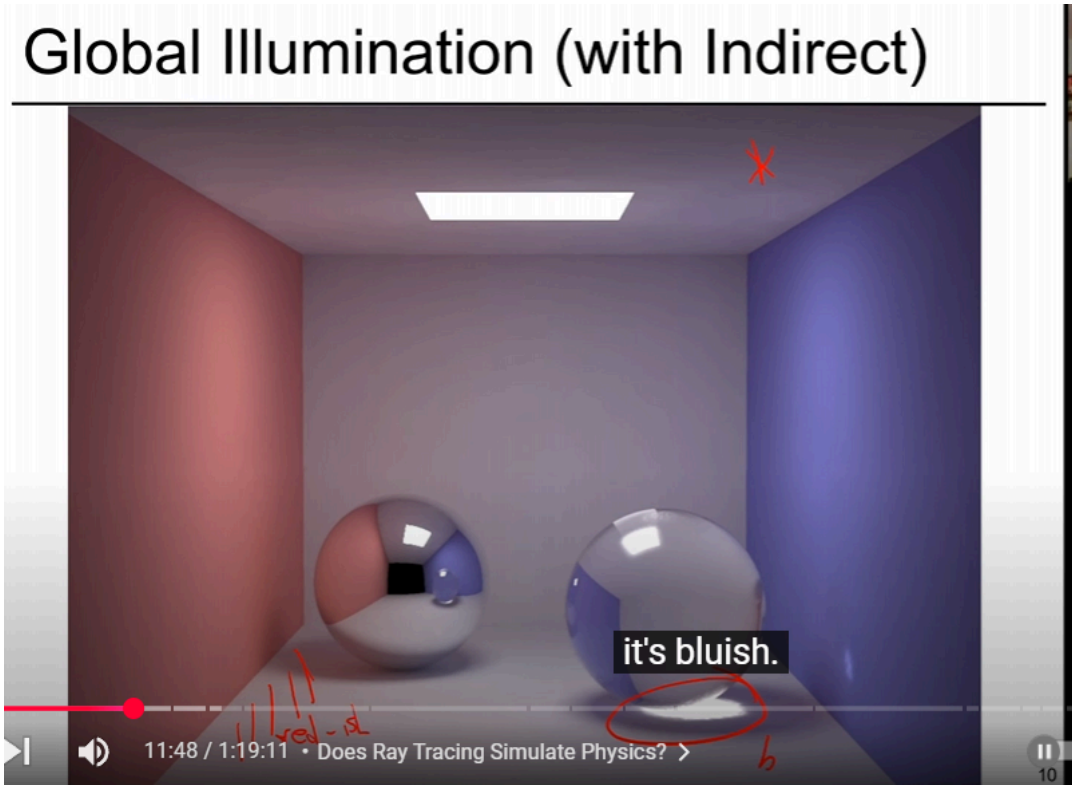
- Example:
    - Current Ray Tracing (Direction Light)



Direct Illumination

Cornell box

    - Global Illumination (Indrect Lighting)

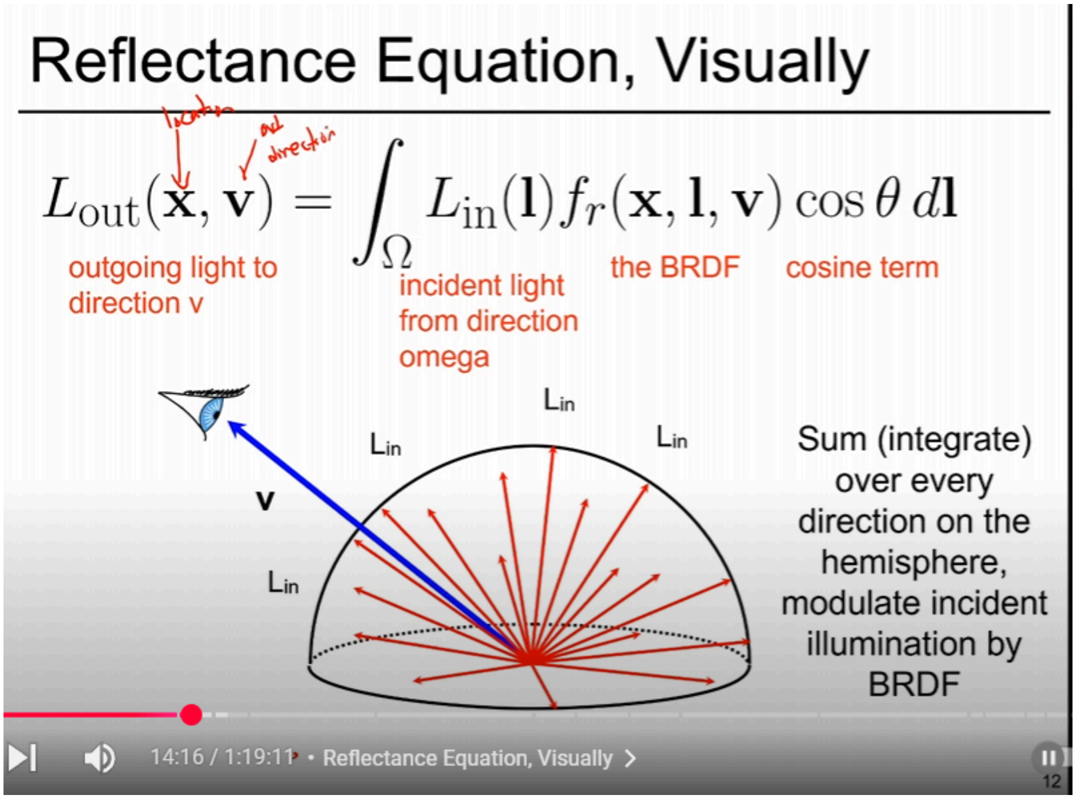


Global Illumination (with Indirect)

it's bluish.

- Rendering Equation

- Reflectance Equation



Reflectance Equation, Visually

$$L_{\text{out}}(\overset{\text{location}}{\mathbf{x}}, \overset{\text{and direction}}{\mathbf{v}}) = \int_{\Omega} L_{\text{in}}(\mathbf{l}) f_r(\mathbf{x}, \mathbf{l}, \mathbf{v}) \cos\theta \, d\mathbf{l}$$

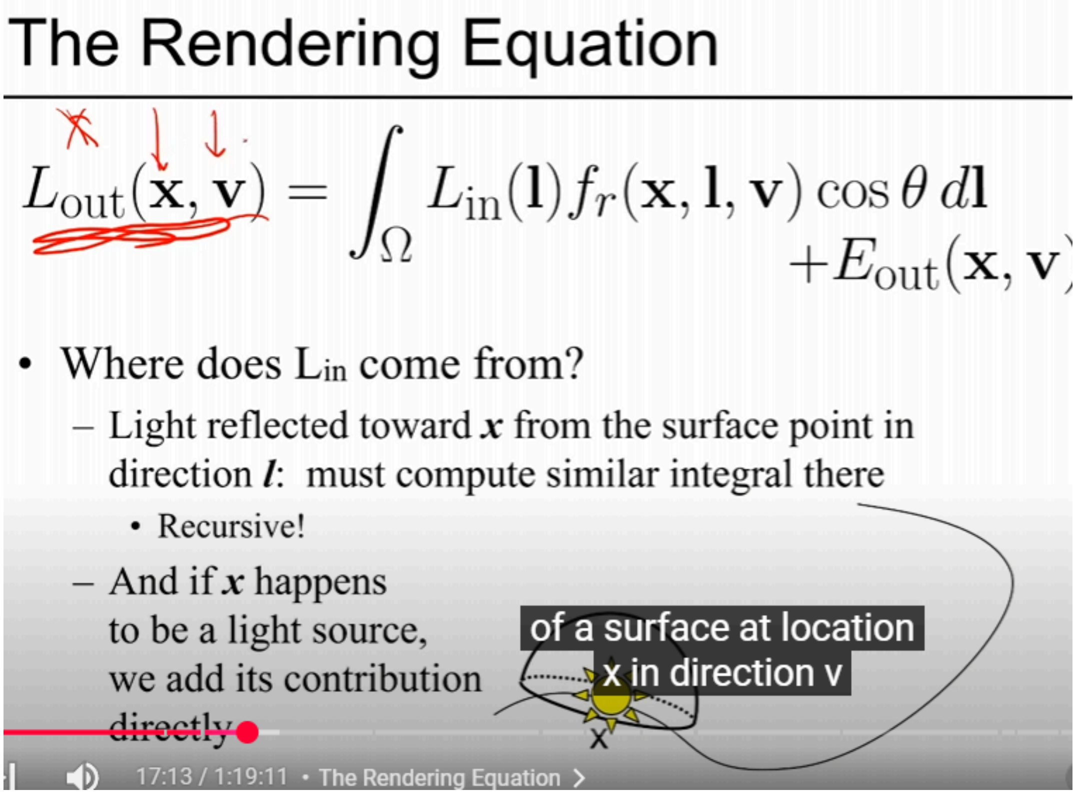
outgoing light to direction v

incident light from direction omega

the BRDF

cosine term

Sum (integrate) over every direction on the hemisphere, modulate incident illumination by BRDF



The Rendering Equation

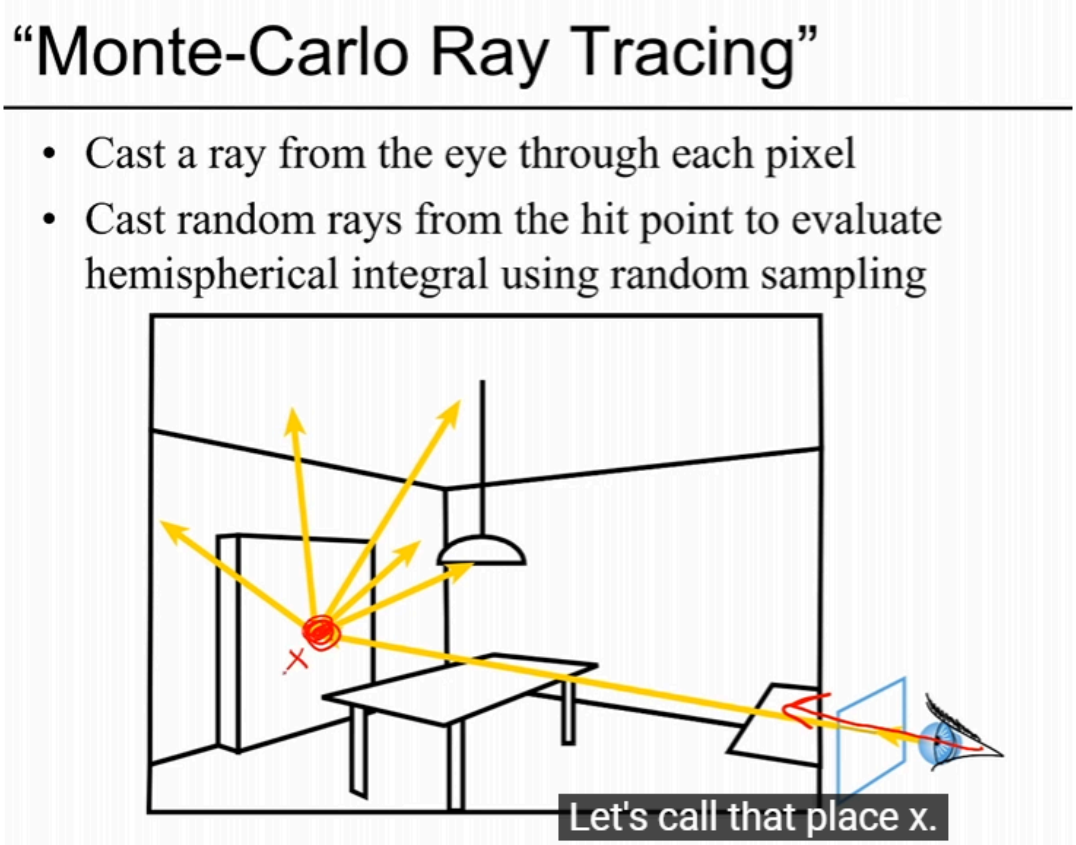
$$L_{\text{out}}(\mathbf{x}, \mathbf{v}) = \int_{\Omega} L_{\text{in}}(\mathbf{l}) f_r(\mathbf{x}, \mathbf{l}, \mathbf{v}) \cos\theta \, d\mathbf{l} + E_{\text{out}}(\mathbf{x}, \mathbf{v})$$

- Where does L$_{\text{in}}$ come from?
  - Light reflected toward ***x*** from the surface point in direction ***l***: must compute similar integral there
    - Recursive!
  - And if ***x*** happens to be a light source, we add its contribution directly

of a surface at location x in direction v

- Path Tracing
- Monte Carlo intergration

- Monte-Carlo Ray Tracing

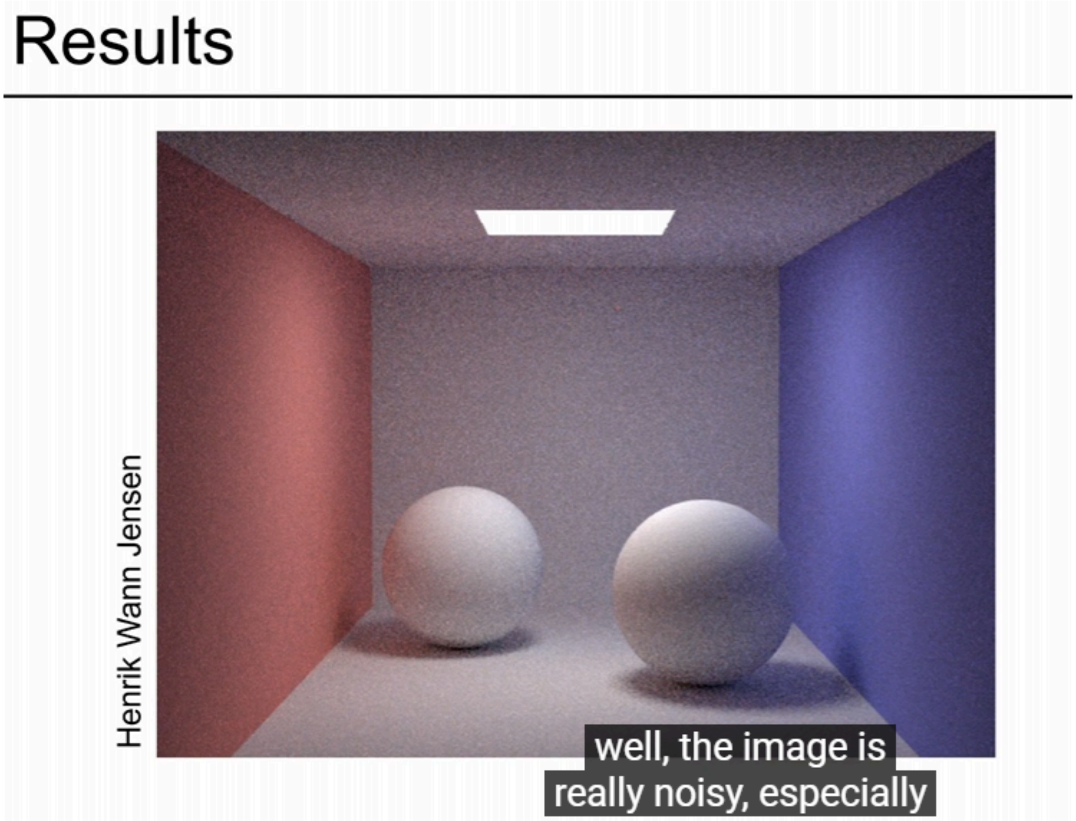


“Monte-Carlo Ray Tracing”

- Cast a ray from the eye through each pixel
- Cast random rays from the hit point to evaluate hemispherical integral using random sampling

Let's call that place x.

- Result



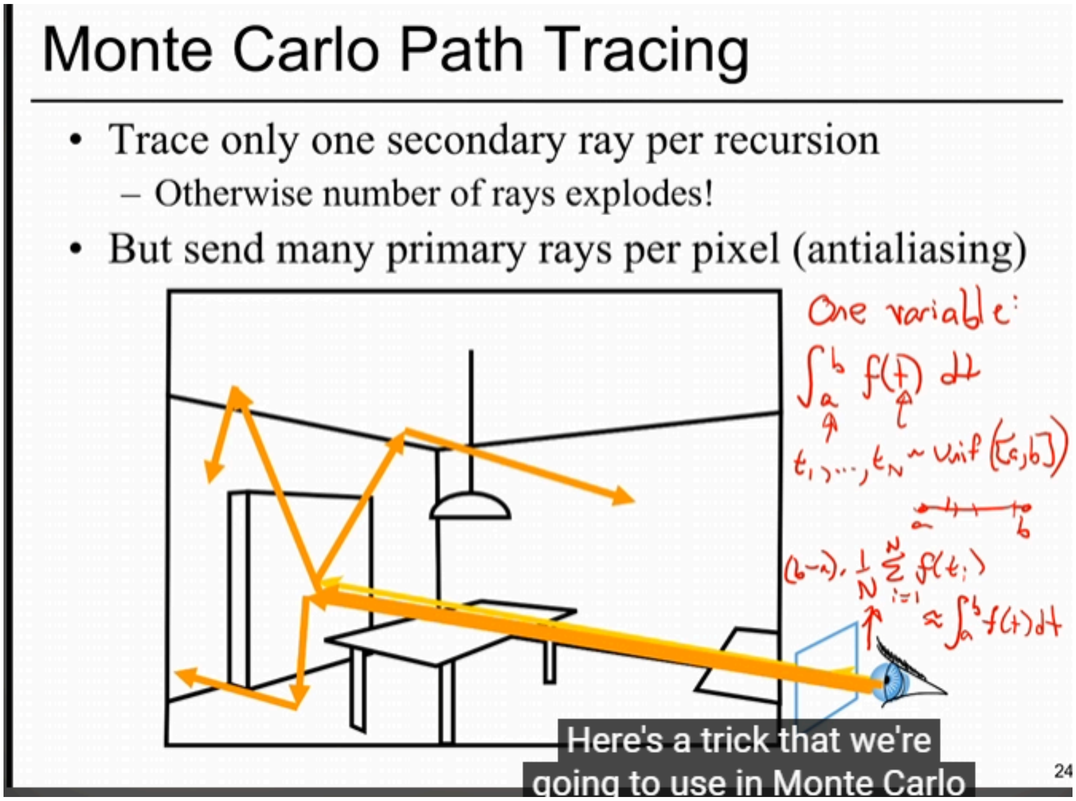
Results

Henrik Wann Jensen

well, the image is really noisy, especially

  - very noisy

- Monte Carlo Path Tracing



Monte Carlo Path Tracing

- Trace only one secondary ray per recursion
  - Otherwise number of rays explodes!
- But send many primary rays per pixel (antialiasing)

One variable:

$\int_a^b f(t)\, dt$

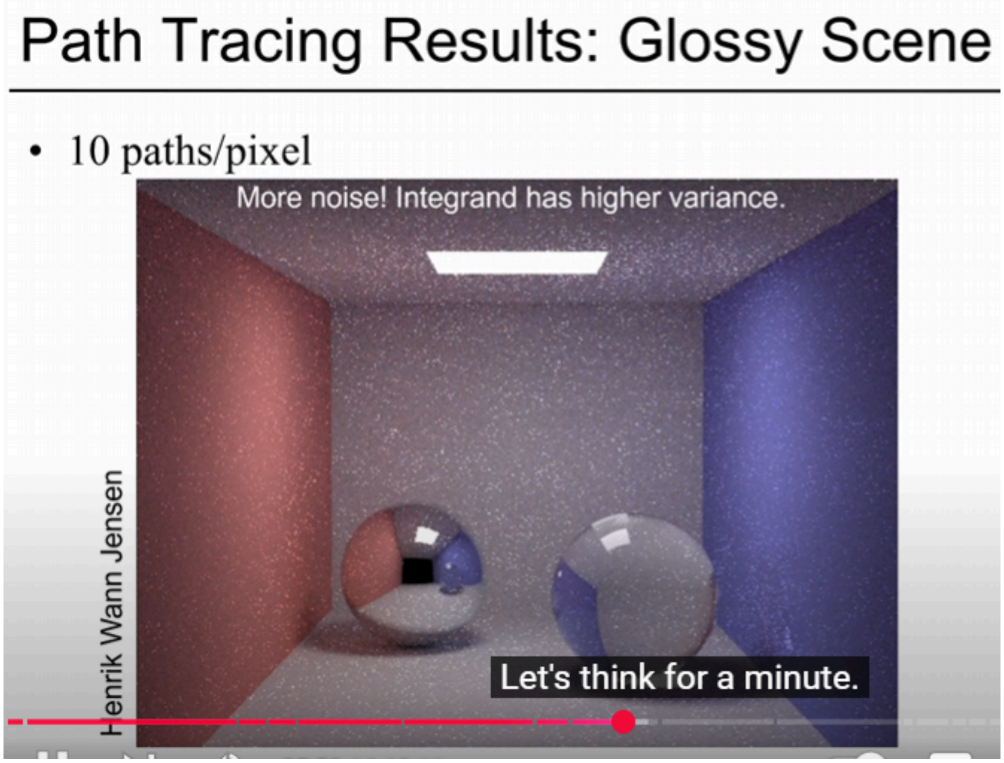
$t_1, \dots, t_N \sim \text{Unif}([a,b])$

$(b-a) \cdot \frac{1}{N} \sum_{i=1}^{N} f(t_i) \approx \int_a^b f(t)\,dt$

Here's a trick that we're going to use in Monte Carlo

24

- Trace only one reflected ray (Random) per time
- And do the Trach Path n times for every pixel, randomize the color
- 10 paths/pixel



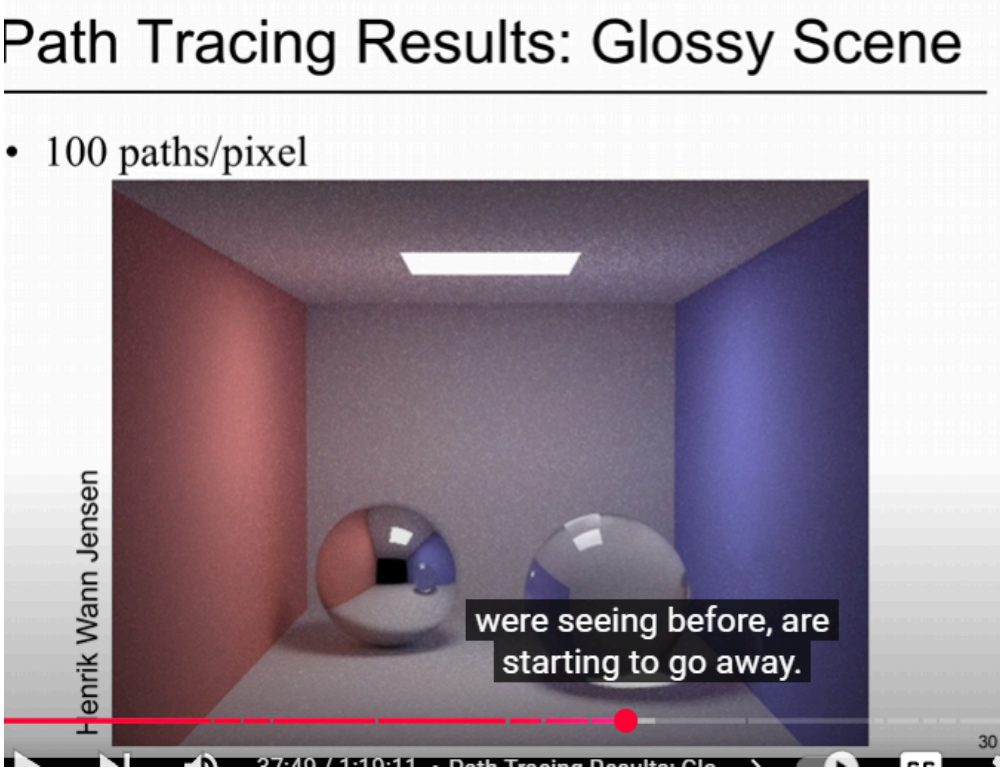
Path Tracing Results: Glossy Scene

- 10 paths/pixel

More noise! Integrand has higher variance.

Let's think for a minute.

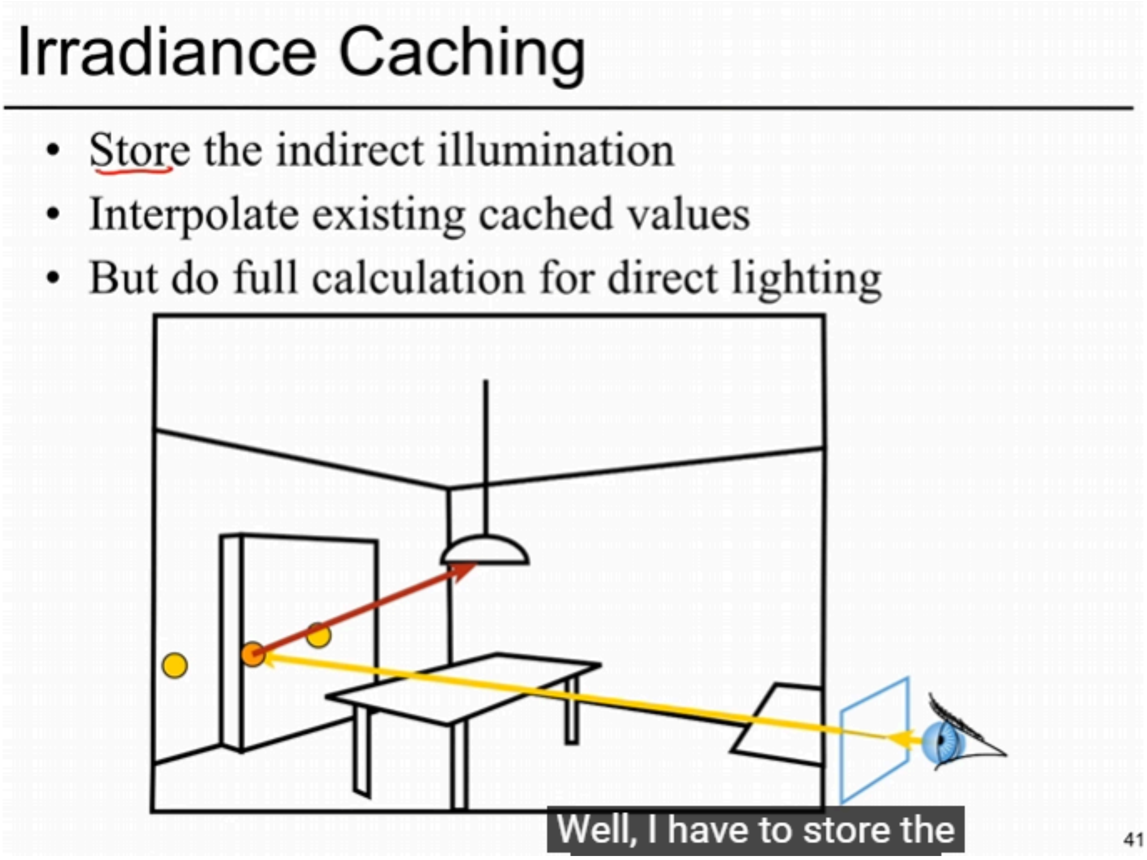
    - 100 paths/pixel



Path Tracing Results: Glossy Scene

- 100 paths/pixel

- Irradiance Caching

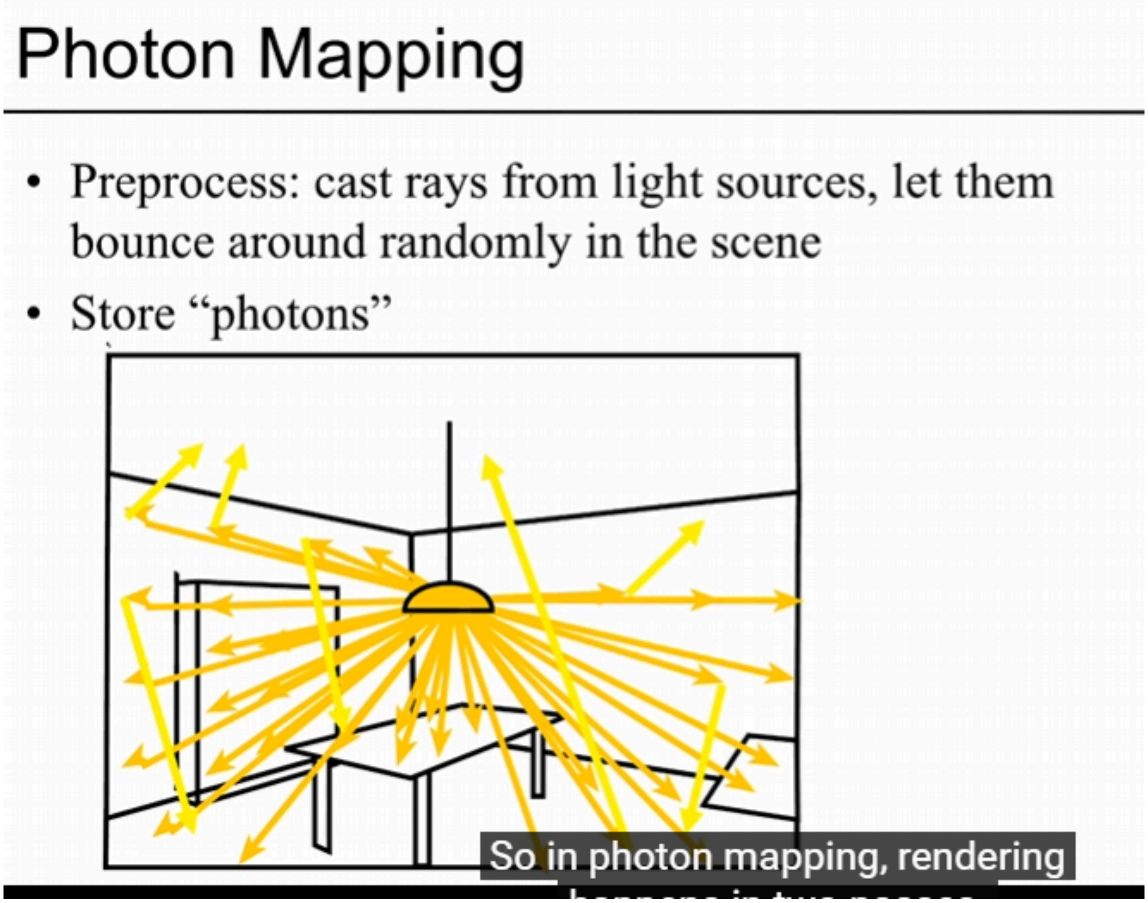


Irradiance Caching

- Store the indirect illumination
- Interpolate existing cached values
- But do full calculation for direct lighting

  - for better optimization
  - Store the value of that point for nearyby usage

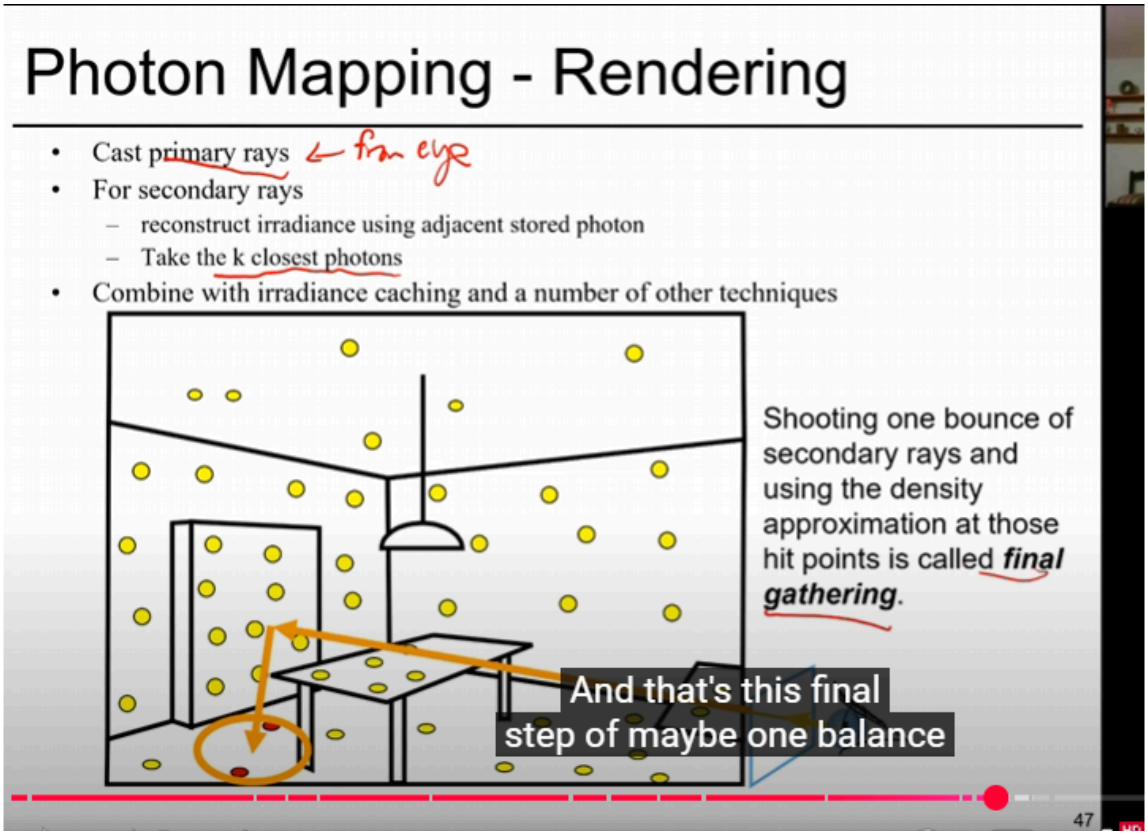
- Photon Mapping



Photon Mapping

- Preprocess: cast rays from light sources, let them bounce around randomly in the scene
- Store “photons”

So in photon mapping, rendering
happens in two passes



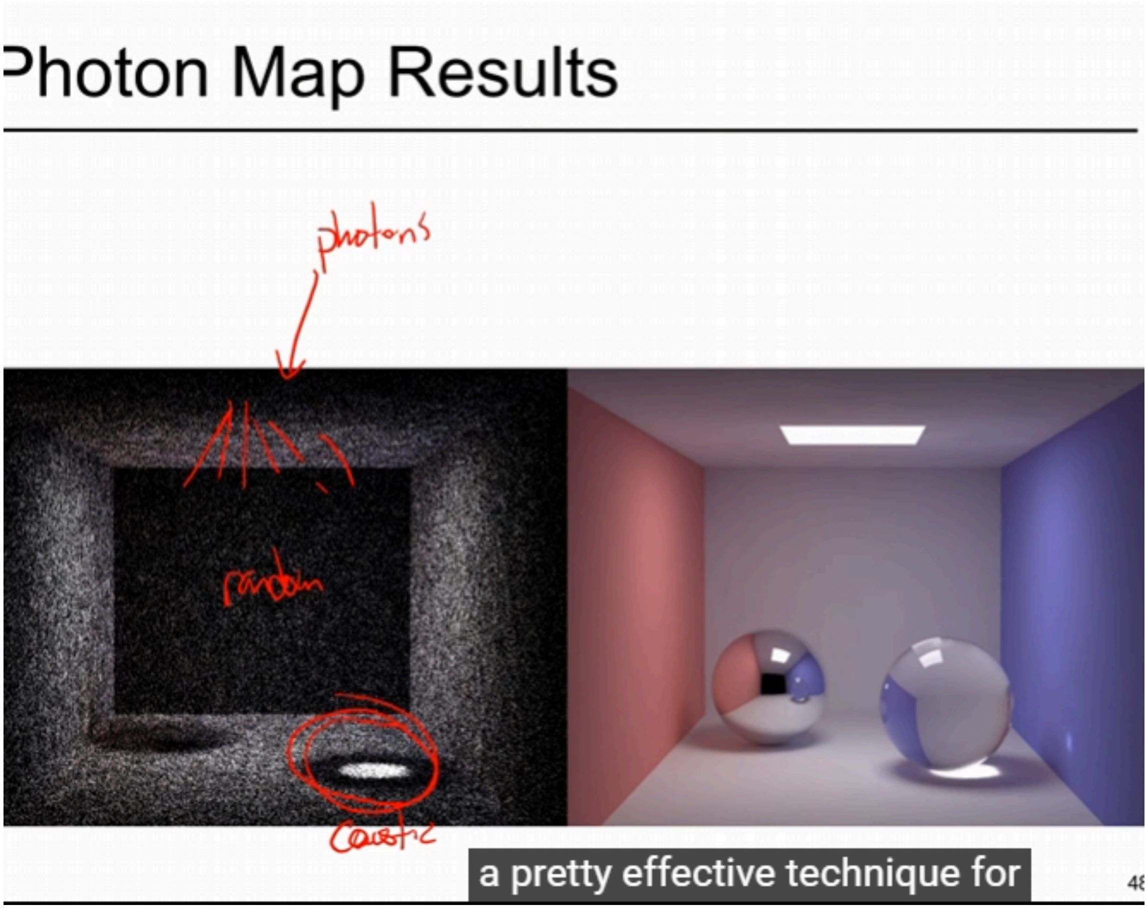
Photon Mapping - Rendering

- Cast primary rays ← from eye
- For secondary rays
  - reconstruct irradiance using adjacent stored photon
  - Take the k closest photons
- Combine with irradiance caching and a number of other techniques

Shooting one bounce of secondary rays and using the density approximation at those hit points is called ***final gathering***.

And that's this final
step of maybe one balance

47



Photon Map Results

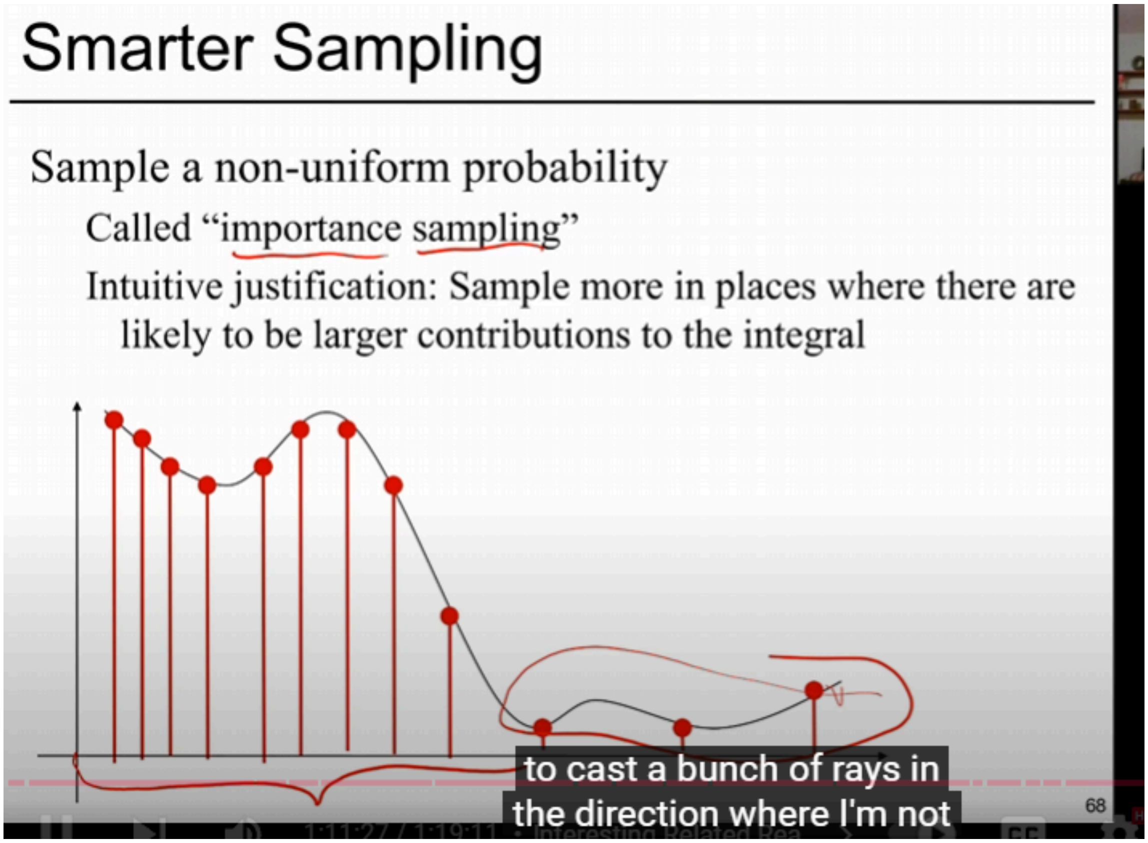
a pretty effective technique for

48

- More Global Illumination

- Ohter Topic: Monte Carlo Integration
  - for average the results

- Better sampling
    - Importance sampling



Smarter Sampling

Sample a non-uniform probability

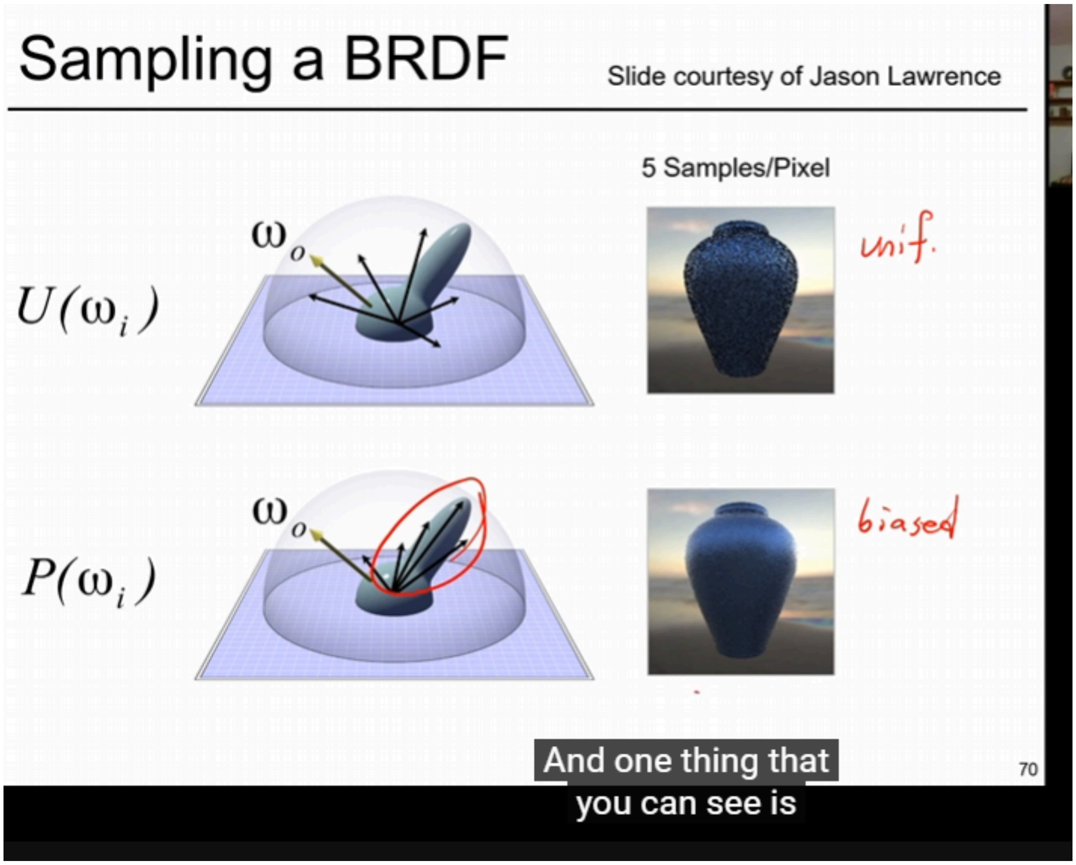
Called “importance sampling”

Intuitive justification: Sample more in places where there are likely to be larger contributions to the integral

to cast a bunch of rays in the direction where I'm not

68

    - biased sampling
    - More Sampling at more lighing area



Sampling a BRDF

Slide courtesy of Jason Lawrence

5 Samples/Pixel

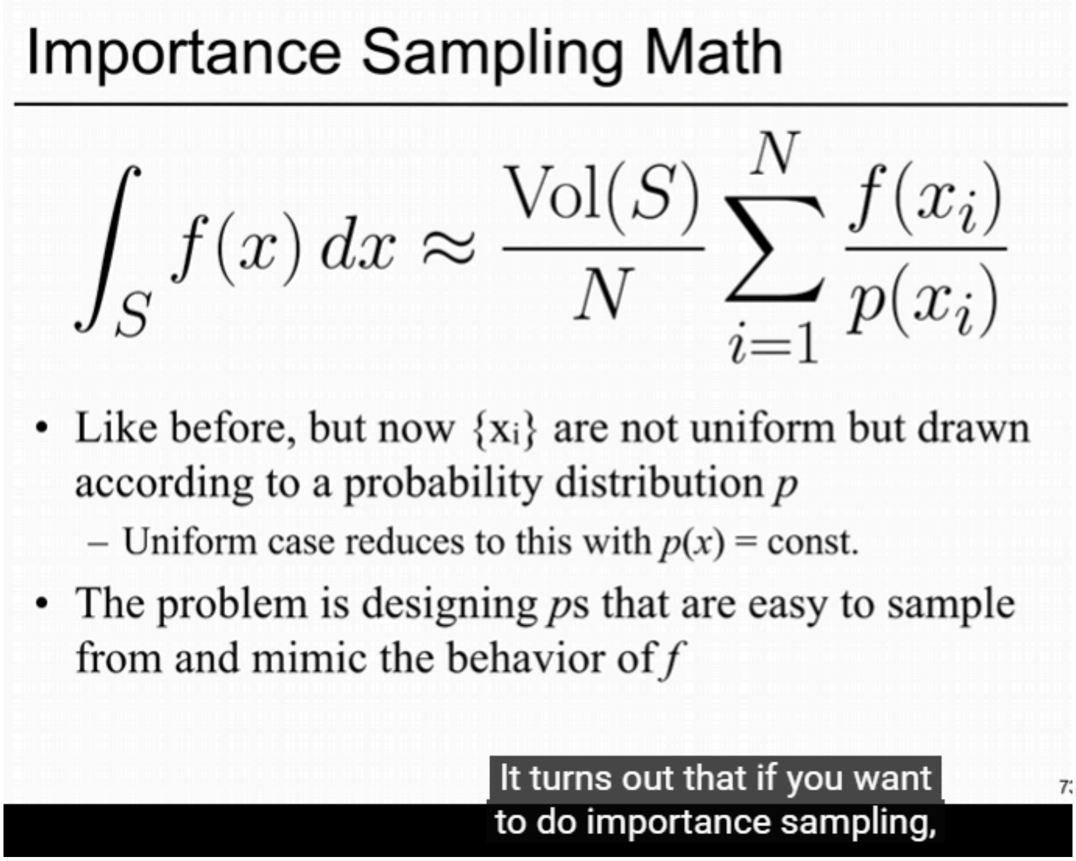
$U(\omega_i)$

$P(\omega_i)$

And one thing that you can see is

70

- Math



Importance Sampling Math

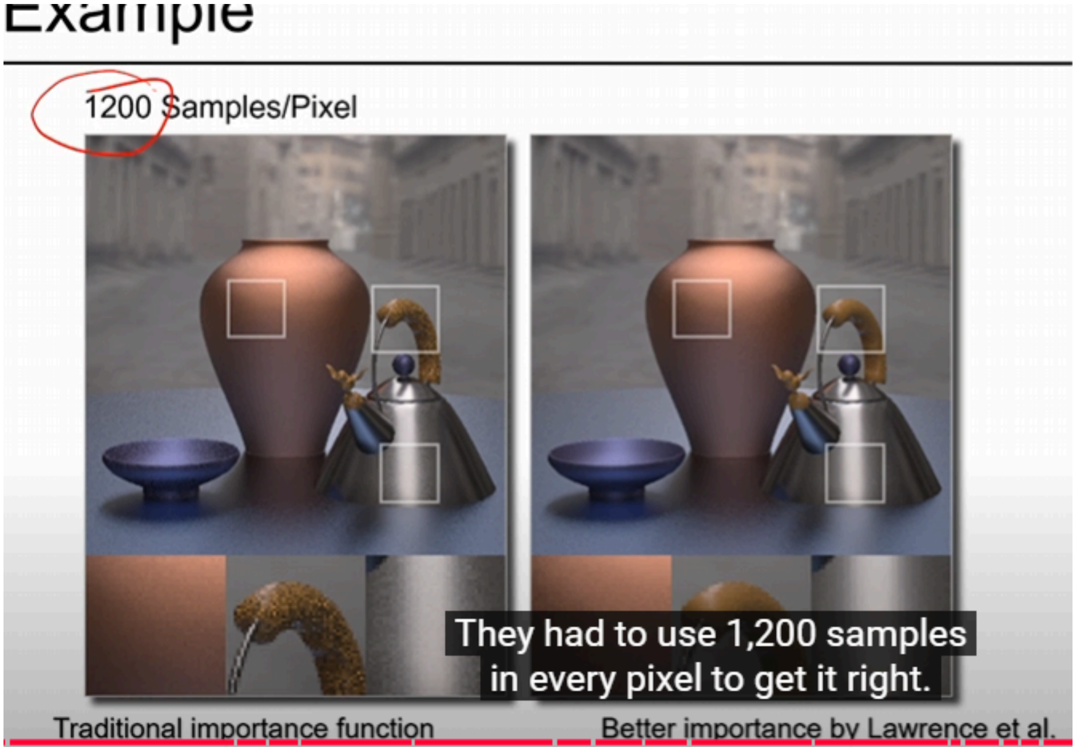
$$\int_S f(x)\,dx \approx \frac{\mathrm{Vol}(S)}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}$$

- Like before, but now $\{x_i\}$ are not uniform but drawn according to a probability distribution *p*
  - Uniform case reduces to this with $p(x) = \text{const.}$
- The problem is designing *p*s that are easy to sample from and mimic the behavior of *f*

It turns out that if you want
to do importance sampling,

  - Divede by the likelihood p(xi)
  - High propability (for sampling) gonna be low weight because it gonna be averaged together in small space

- Example



Example

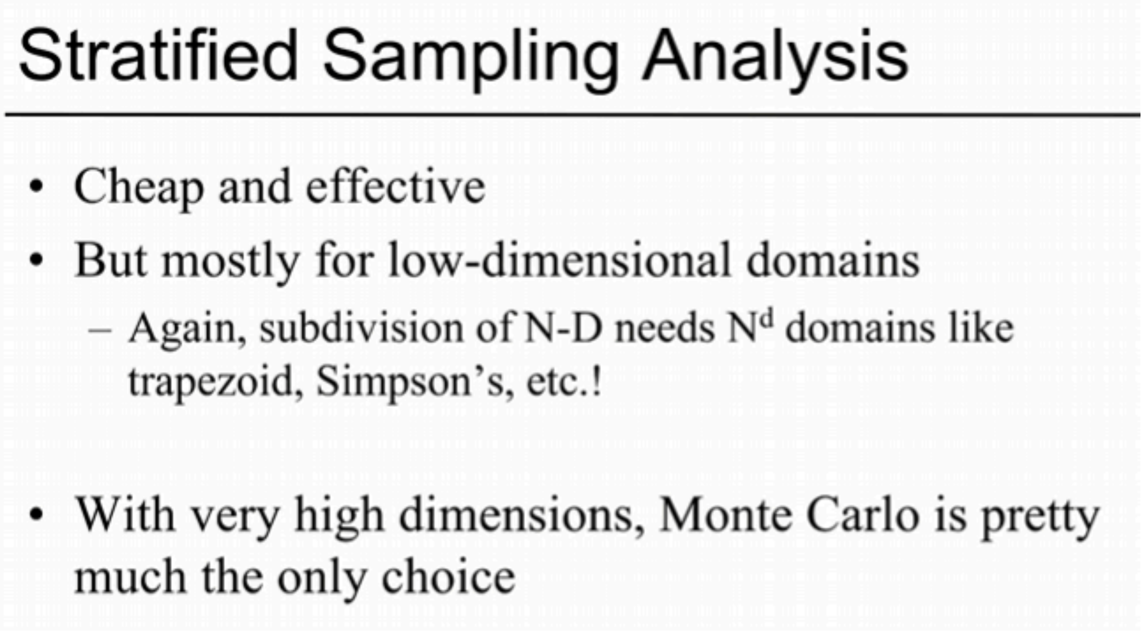
1200 Samples/Pixel

They had to use 1,200 samples
in every pixel to get it right.

Traditional importance function

Better importance by Lawrence et al.

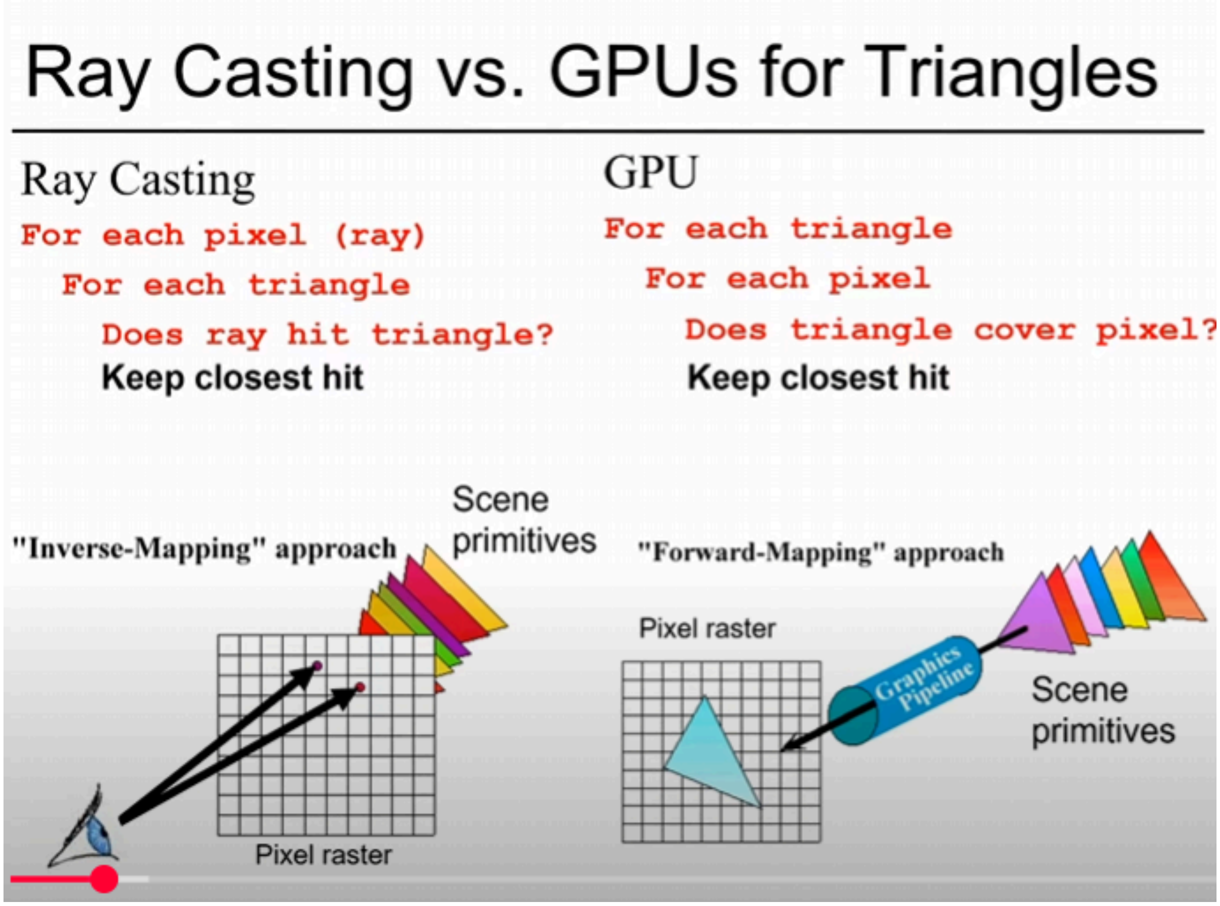
    - Stratification



Stratified Sampling Analysis

- Cheap and effective
- But mostly for low-dimensional domains
  - Again, subdivision of N-D needs N^d domains like trapezoid, Simpson's, etc.!
- With very high dimensions, Monte Carlo is pretty much the only choice

- **L17: Rasterization**
  - Ray Casting vs. GPUs for triangles



Ray Casting vs. GPUs for Triangles

Ray Casting

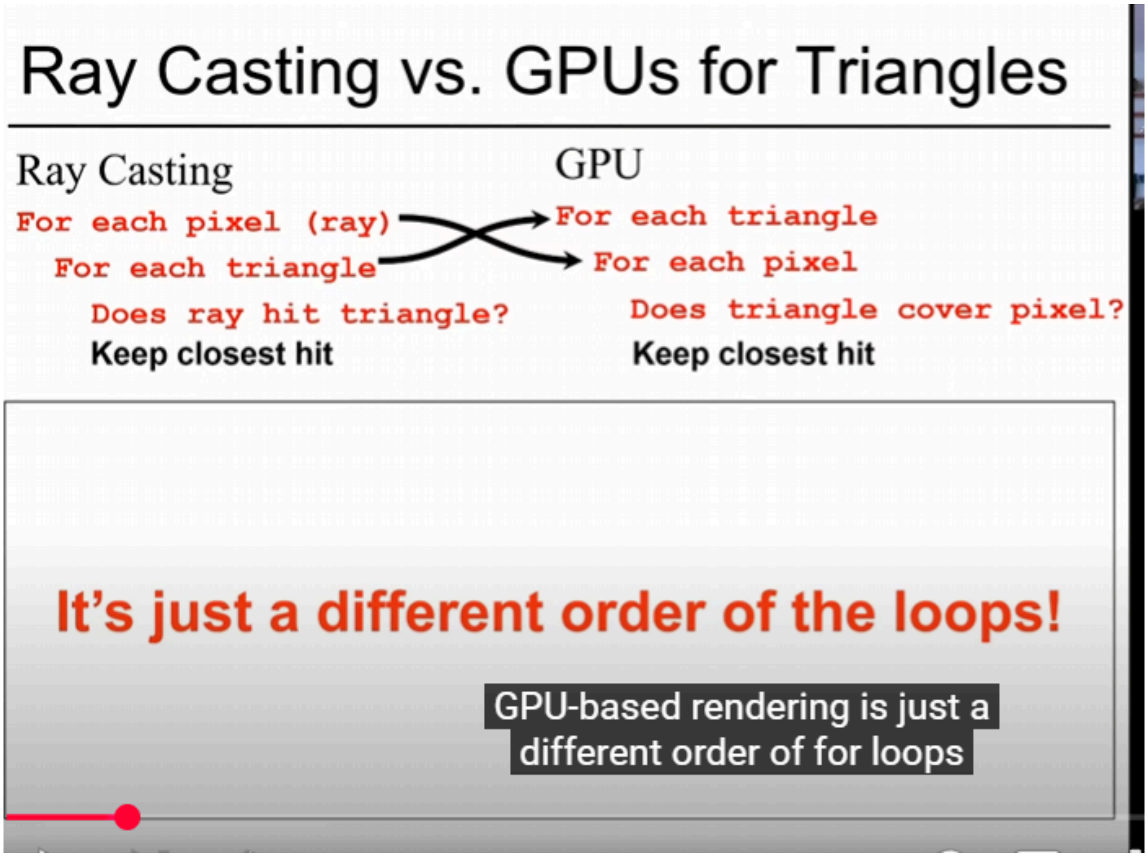
```
For each pixel (ray)
    For each triangle
        Does ray hit triangle?
        Keep closest hit
```

GPU

```
For each triangle
    For each pixel
        Does triangle cover pixel?
        Keep closest hit
```

    - Ray casting
      - Draw 1 pixel at a time
    - GPU
      - Draw 1 trangle at a time

- Different Order



Ray Casting vs. GPUs for Triangles

Ray Casting

```
For each pixel (ray)
  For each triangle
    Does ray hit triangle?
    Keep closest hit
```

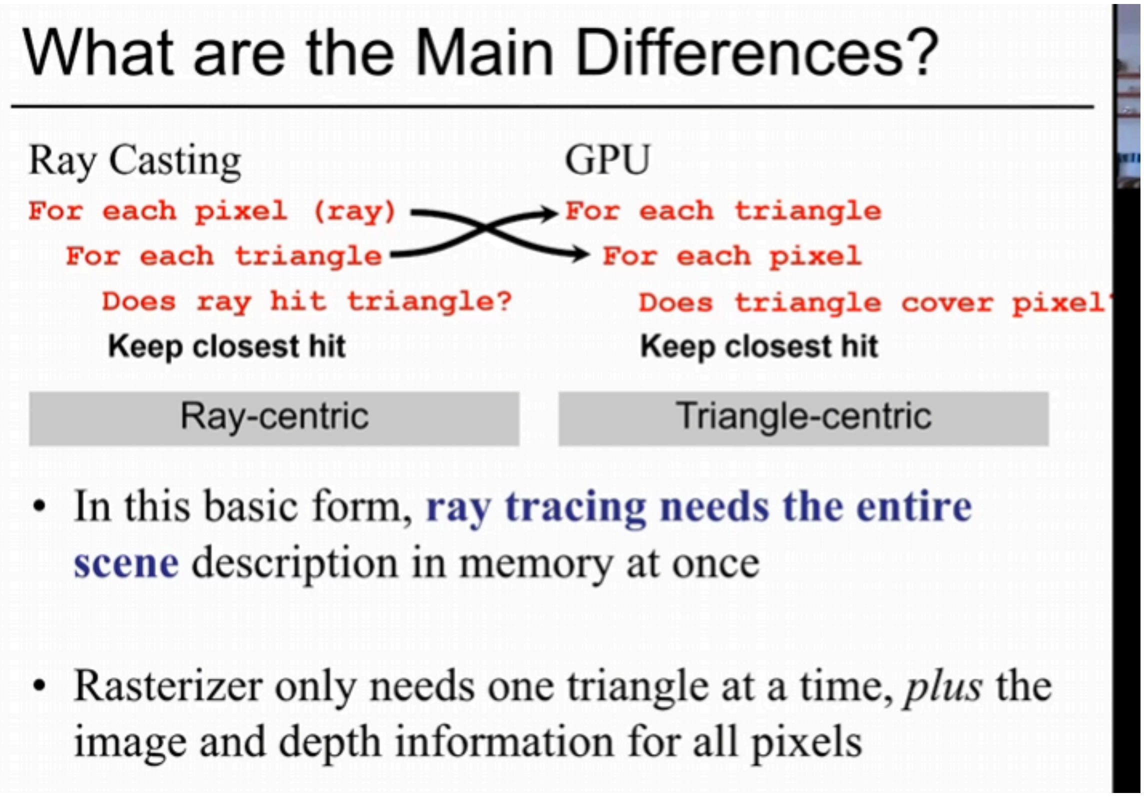
GPU

```
For each triangle
  For each pixel
    Does triangle cover pixel?
    Keep closest hit
```

It's just a different order of the loops!

GPU-based rendering is just a different order of for loops

- Main Difference



What are the Main Differences?

Ray Casting

```
For each pixel (ray)
  For each triangle
    Does ray hit triangle?
    Keep closest hit
```

GPU

```
For each triangle
  For each pixel
    Does triangle cover pixel
    Keep closest hit
```

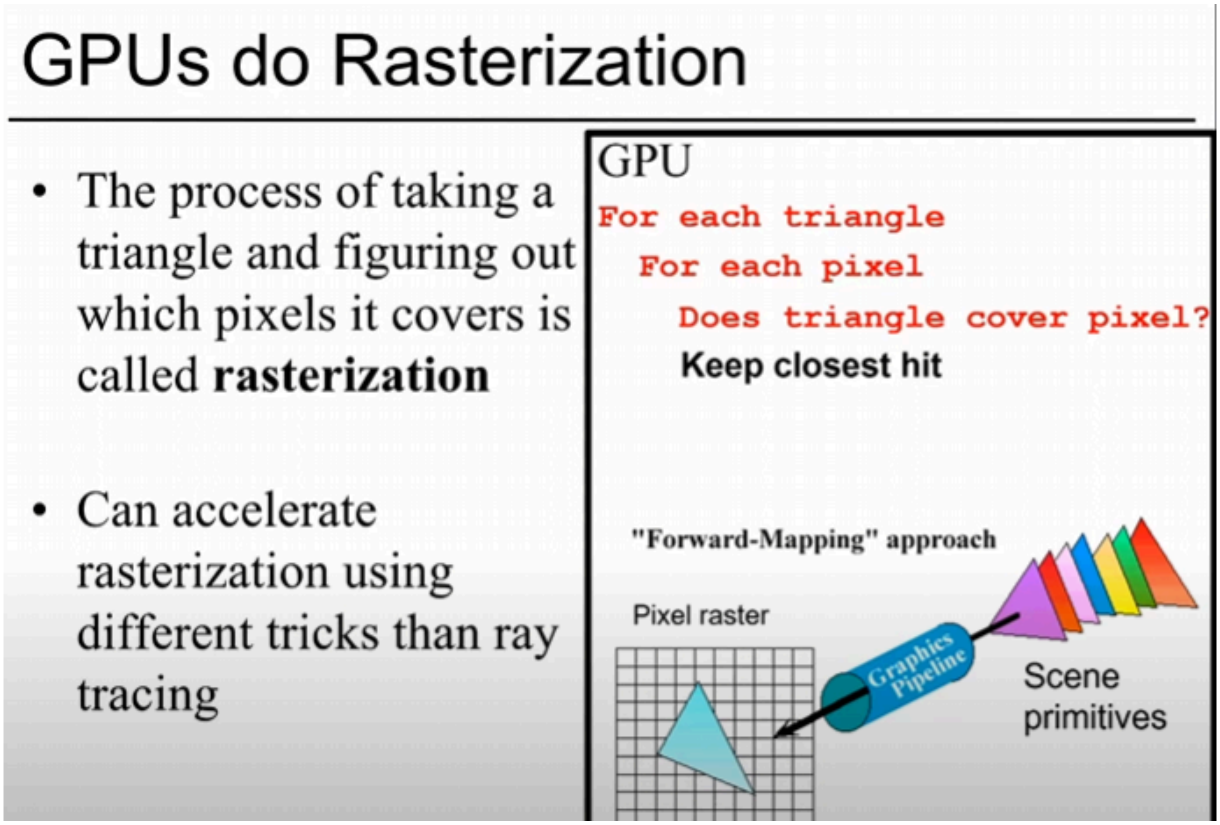
Ray-centric

Triangle-centric

- In this basic form, **ray tracing needs the entire scene** description in memory at once
- Rasterizer only needs one triangle at a time, *plus* the image and depth information for all pixels

  - Ray tracing need the entire scene in memory
  - Rasterizer only need one triangle at a time, and the image and depth
  - Rasterization use less memory

- GPU Rasterization Overview



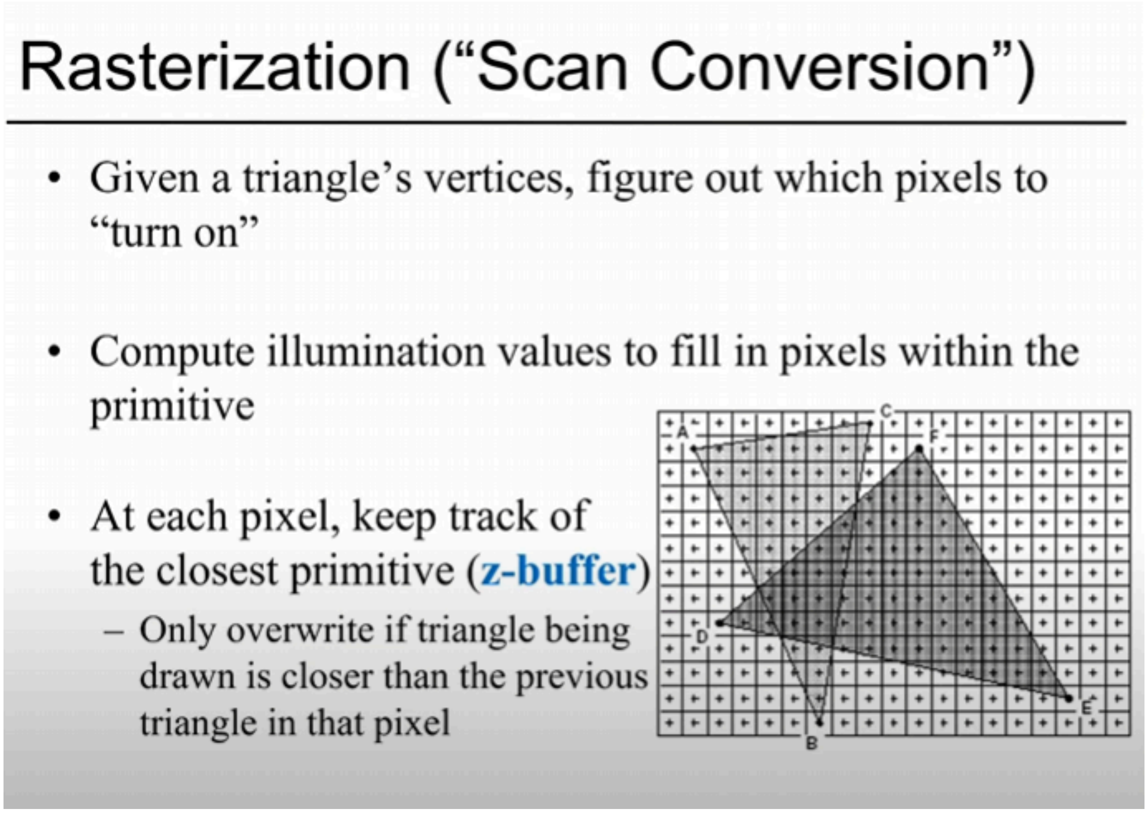
GPUs do Rasterization

- The process of taking a triangle and figuring out which pixels it covers is called **rasterization**
- Can accelerate rasterization using different tricks than ray tracing

GPU

```
For each triangle
  For each pixel
    Does triangle cover pixel?
    Keep closest hit
```

  - What rasterization actually do (Scan Conversion)

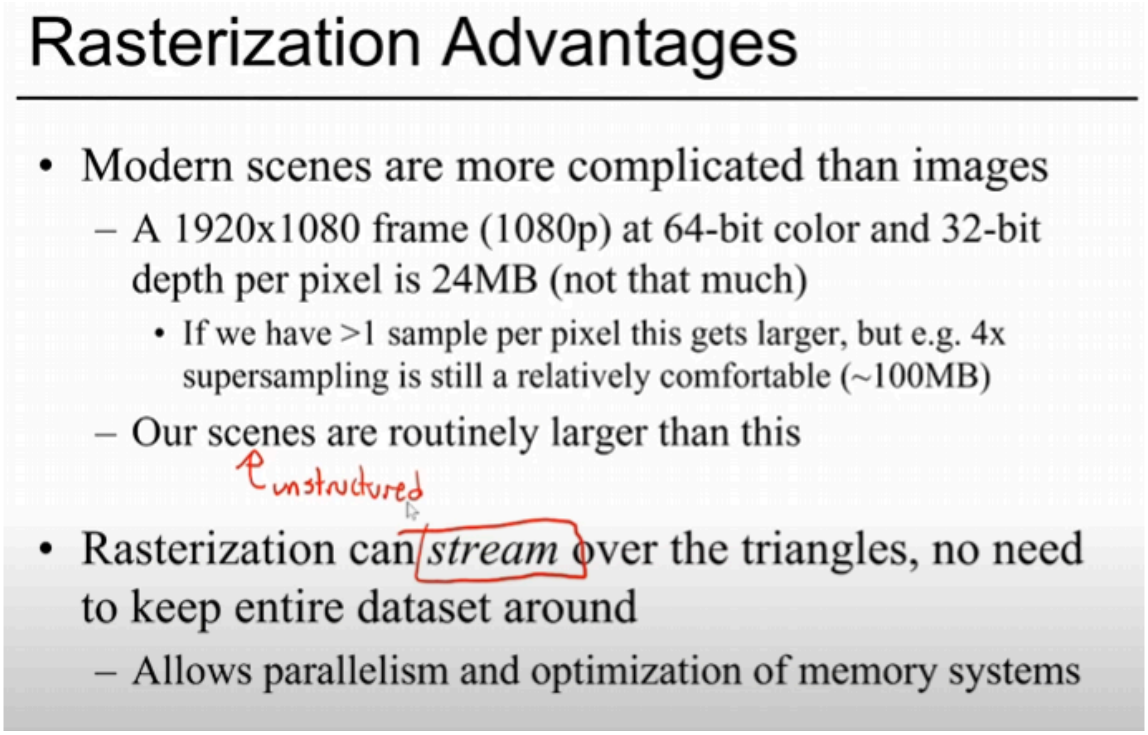


Rasterization ("Scan Conversion")

- Given a triangle's vertices, figure out which pixels to "turn on"
- Compute illumination values to fill in pixels within the primitive
- At each pixel, keep track of the closest primitive (**z-buffer**)
  - Only overwrite if triangle being drawn is closer than the previous triangle in that pixel

    - z-buffer
      - determine the depth of the traingle, only show the closest one

- Rasterization Pros



Rasterization Advantages

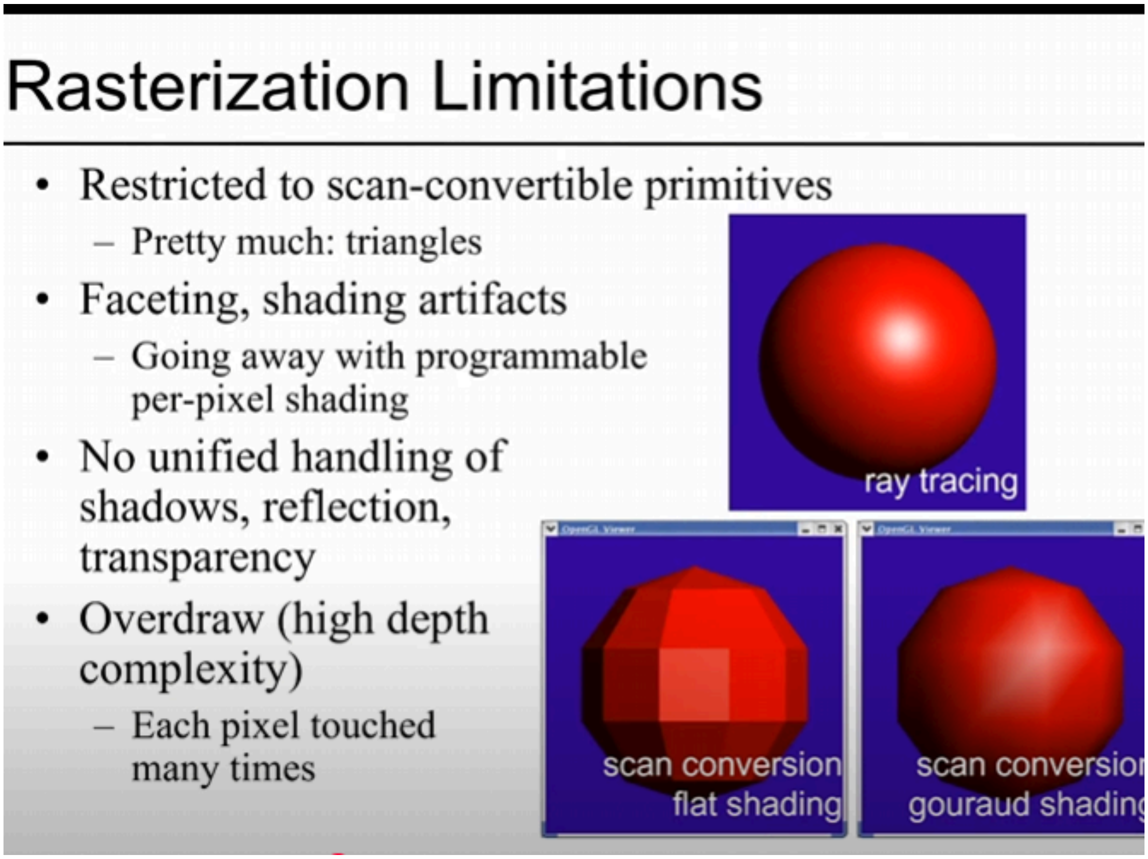
- Modern scenes are more complicated than images
  - A 1920x1080 frame (1080p) at 64-bit color and 32-bit depth per pixel is 24MB (not that much)
    - If we have >1 sample per pixel this gets larger, but e.g. 4x supersampling is still a relatively comfortable (~100MB)
  - Our scenes are routinely larger than this

unstructured

- Rasterization can *stream* over the triangles, no need to keep entire dataset around
  - Allows parallelism and optimization of memory systems

  - use less memory

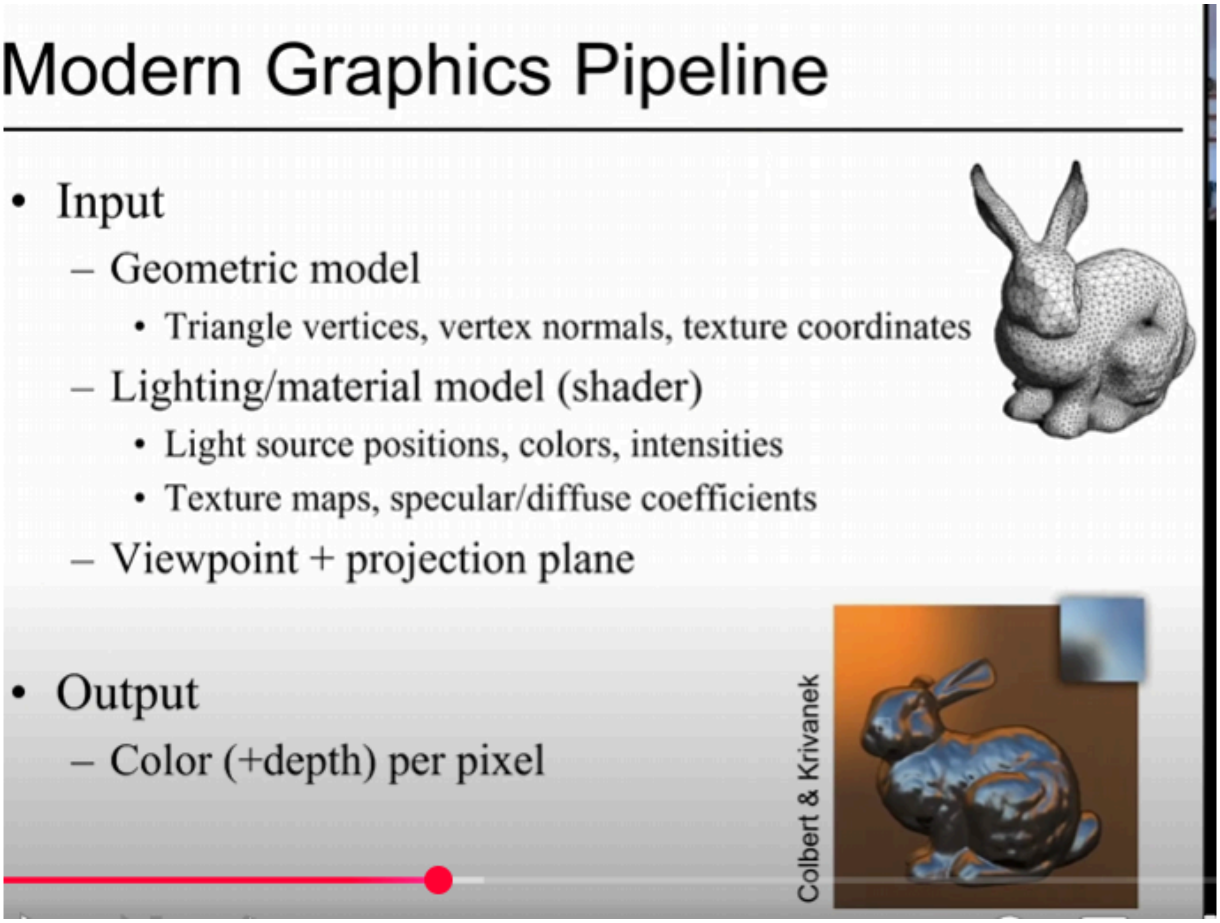
- Rasterization Cons



Rasterization Limitations

- Restricted to scan-convertible primitives
  - Pretty much: triangles
- Faceting, shading artifacts
  - Going away with programmable per-pixel shading
- No unified handling of shadows, reflection, transparency
- Overdraw (high depth complexity)
  - Each pixel touched many times

- Modern Graphics Pipeline

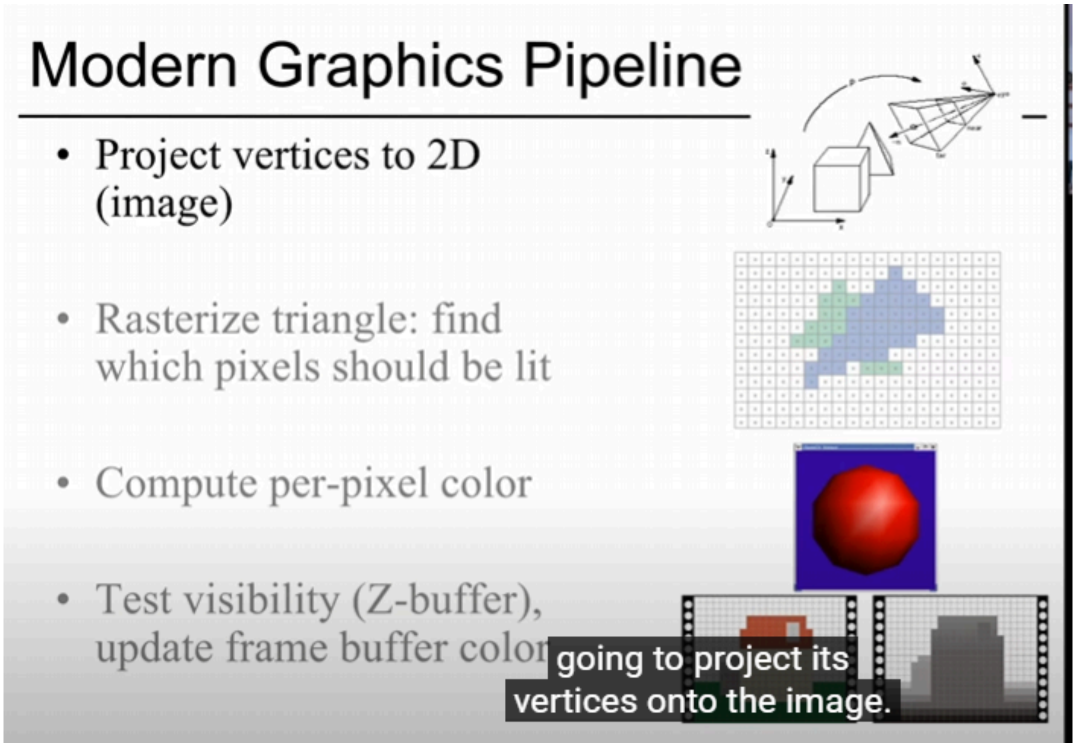


Modern Graphics Pipeline

- Input
  - Geometric model
    - Triangle vertices, vertex normals, texture coordinates
  - Lighting/material model (shader)
    - Light source positions, colors, intensities
    - Texture maps, specular/diffuse coefficients
  - Viewpoint + projection plane
- Output
  - Color (+depth) per pixel

Colbert & Krivanek

  - Procedure
    - Step 1: Project vertices to 2D

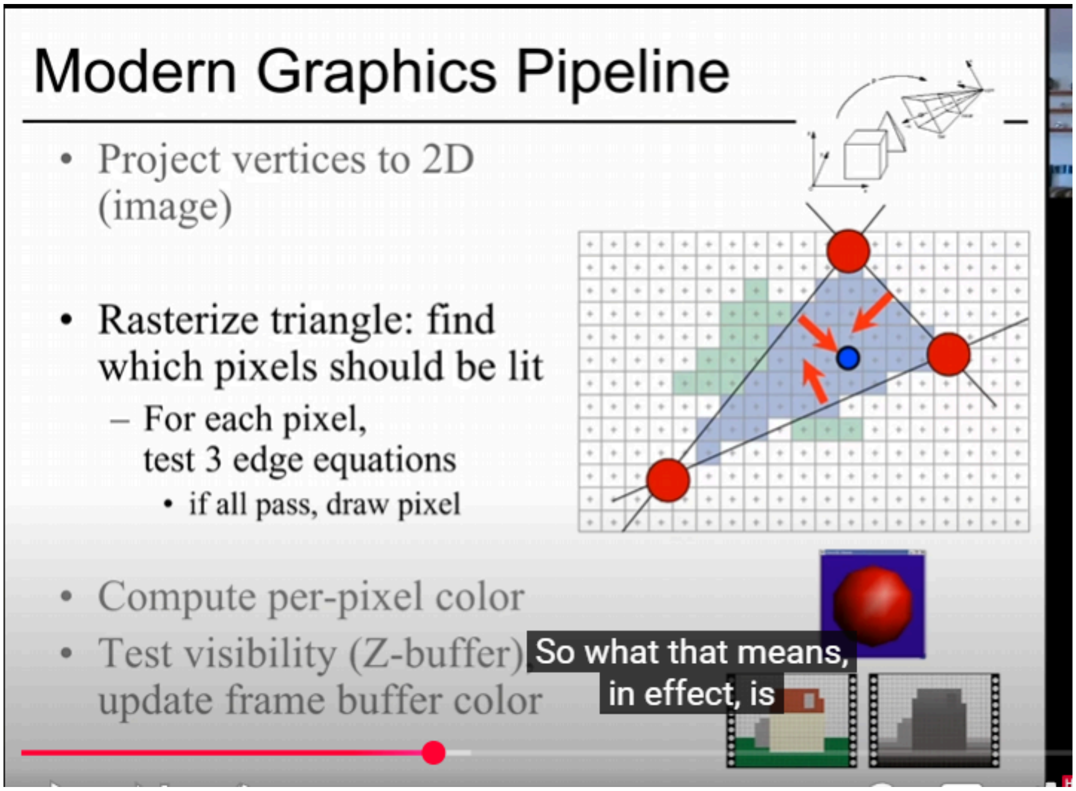


Modern Graphics Pipeline

- Project vertices to 2D (image)
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer color

going to project its vertices onto the image.

    - Step 2: Rasterize triangle: find which pixels shoud be lit

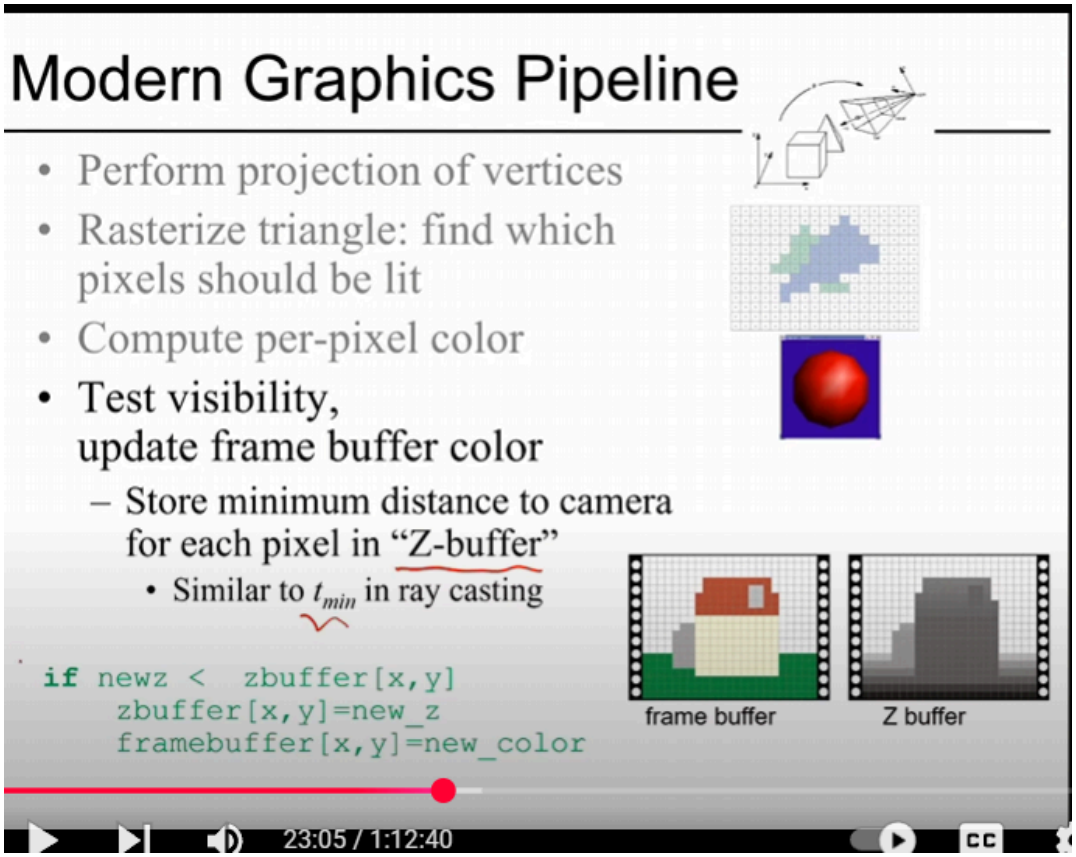


Modern Graphics Pipeline

- Project vertices to 2D (image)
- Rasterize triangle: find which pixels should be lit
  - For each pixel, test 3 edge equations
    - if all pass, draw pixel
- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer color

So what that means, in effect, is

    - Step 3: Compute per-pixel color
    - Step 4: Test visibility, update frame buffer color



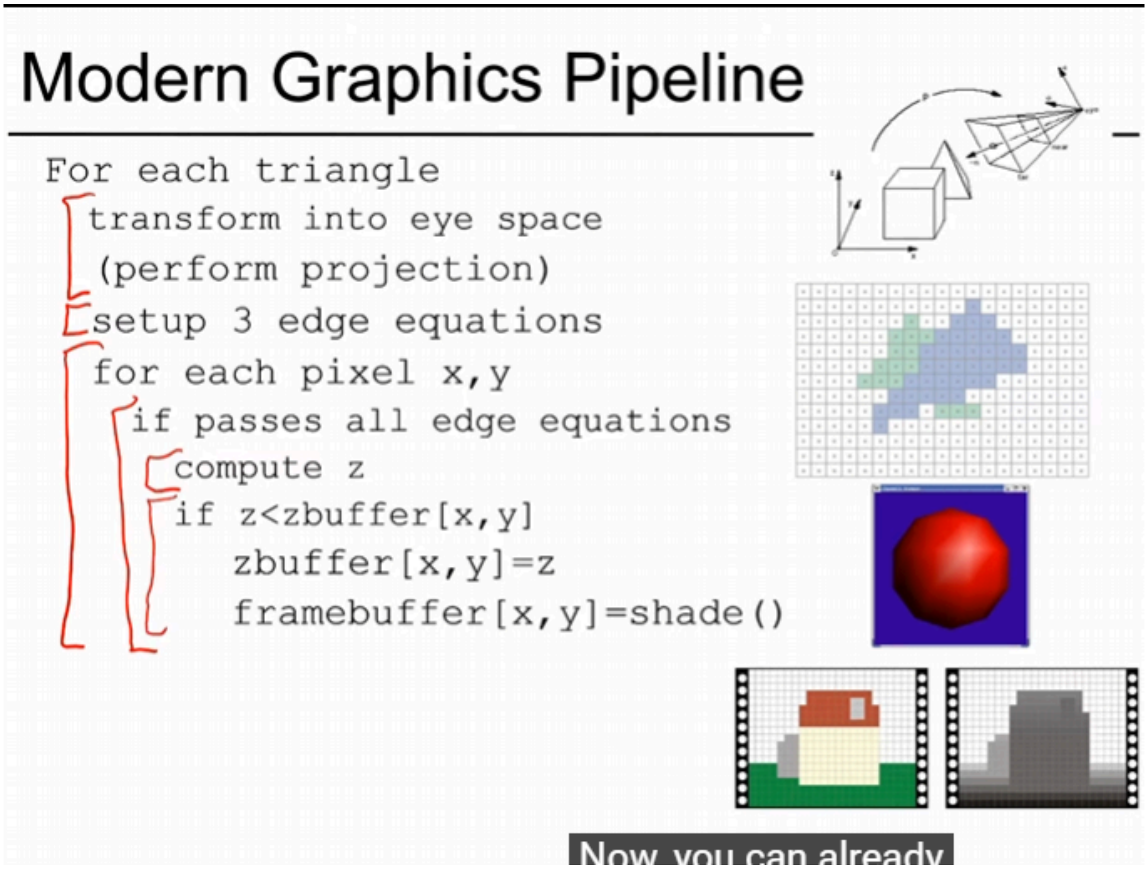
Modern Graphics Pipeline

- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility, update frame buffer color
  - Store minimum distance to camera for each pixel in “Z-buffer”
    - Similar to t_{min} in ray casting

```
if newz <  zbuffer[x,y]
   zbuffer[x,y]=new_z
   framebuffer[x,y]=new_color
```

- Double-buffer

    - show the current frame, prepare the next frame in another buffer, then flip the buffer back and forth.
- Psudo code

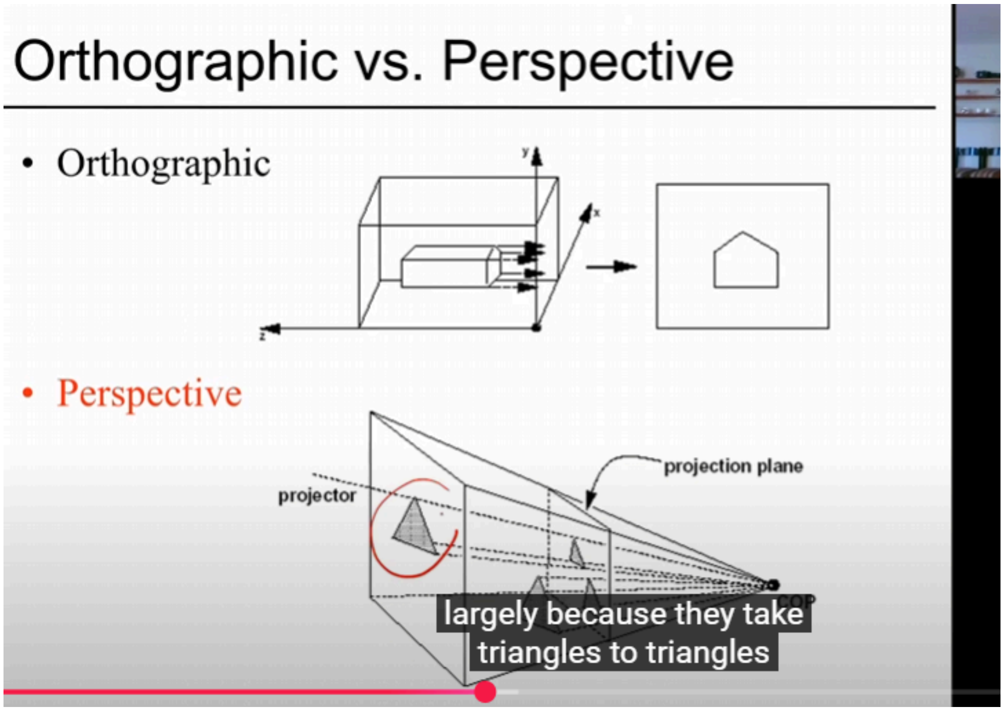


Modern Graphics Pipeline

```
For each triangle
  transform into eye space
   (perform projection)
  setup 3 edge equations
  for each pixel x,y
    if passes all edge equations
      compute z
      if z<zbuffer[x,y]
          zbuffer[x,y]=z
          framebuffer[x,y]=shade()
```

Now you can already

- Step in details
    - Projection vertices to 2D
        - Prthographic vs. Perspective

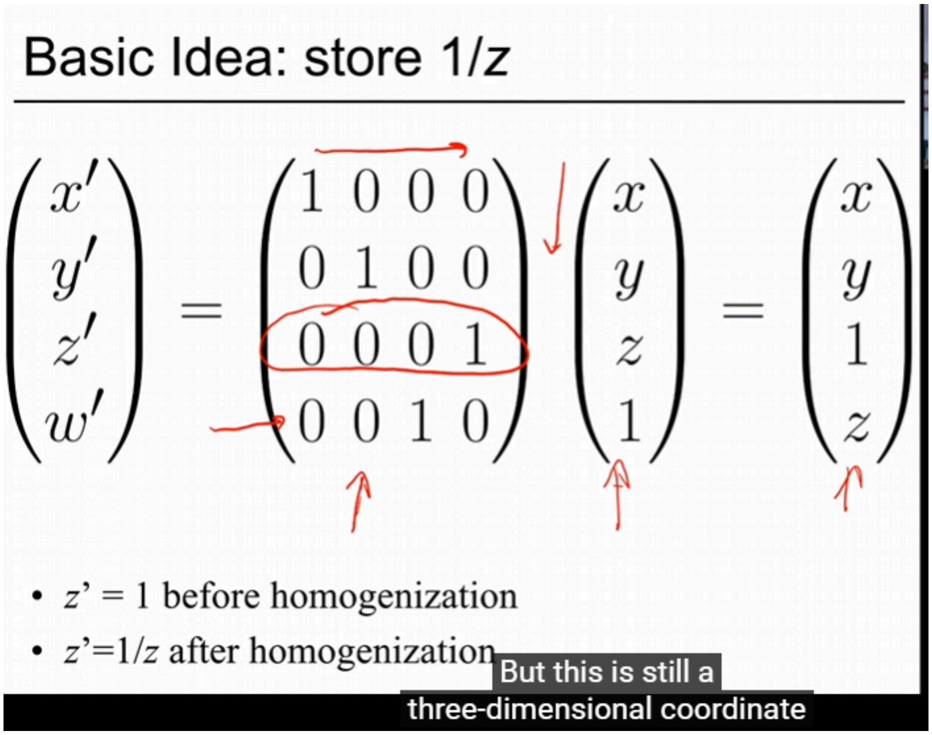


Orthographic vs. Perspective

- Orthographic
- Perspective

largely because they take triangles to triangles

- Perspective

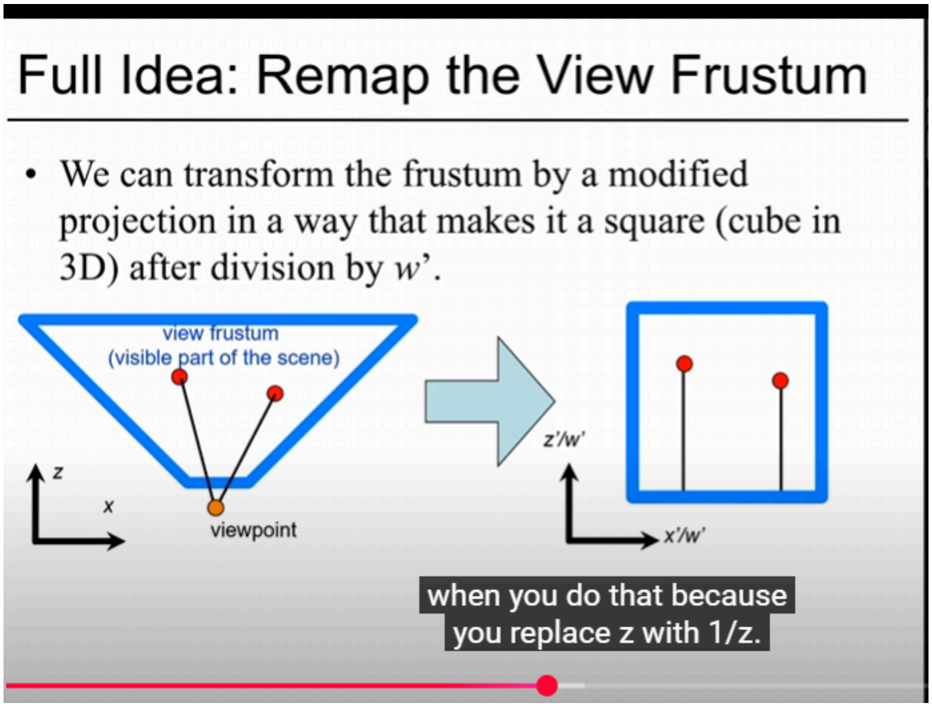


Basic Idea: store 1/z

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ z \end{pmatrix}$$

- $z' = 1$ before homogenization
- $z'=1/z$ after homogenization

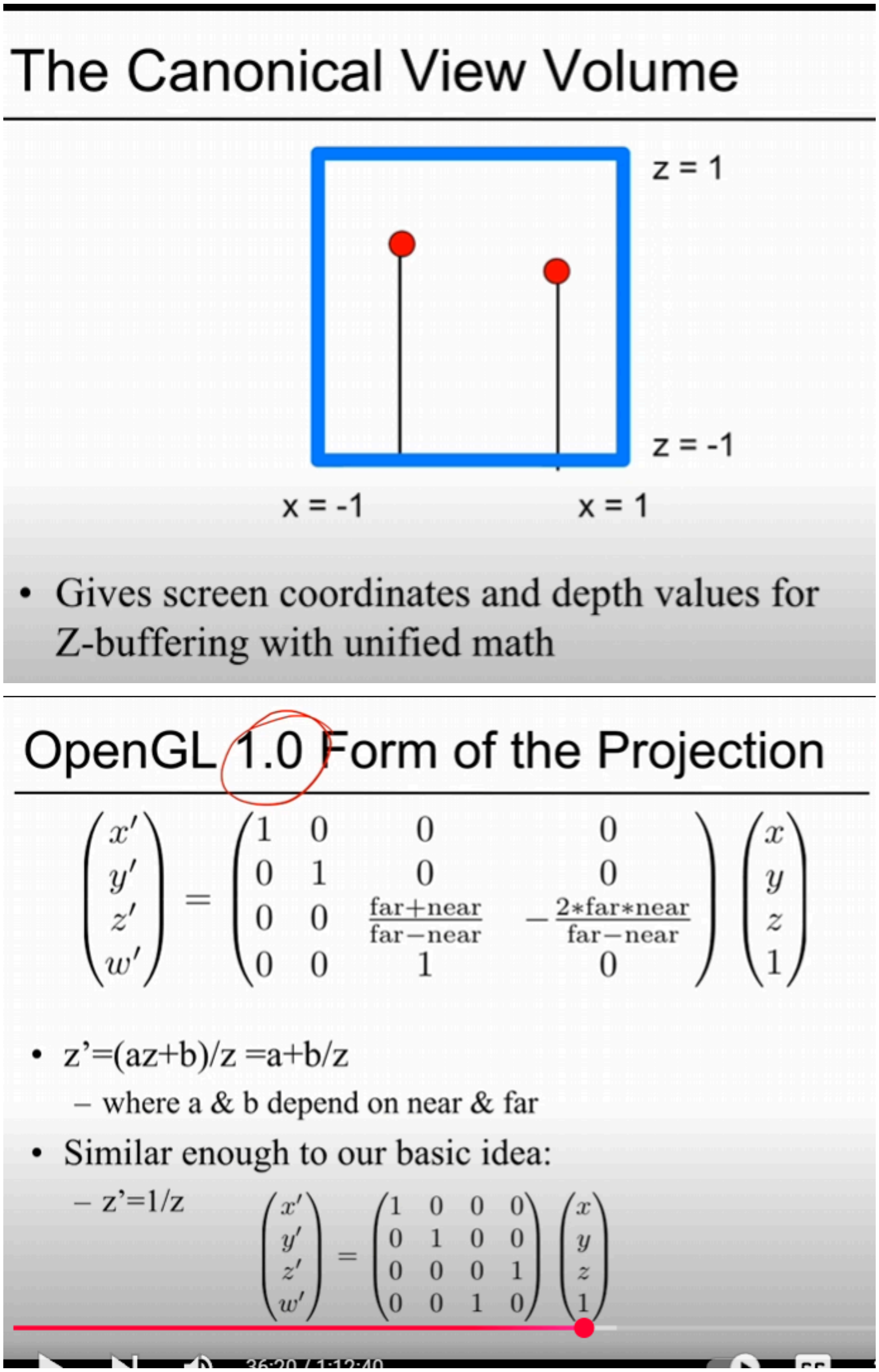
But this is still a
three-dimensional coordinate



Full Idea: Remap the View Frustum

- We can transform the frustum by a modified projection in a way that makes it a square (cube in 3D) after division by *w*'.

when you do that because
you replace z with 1/z.



The Canonical View Volume

- Gives screen coordinates and depth values for Z-buffering with unified math

OpenGL 1.0 Form of the Projection

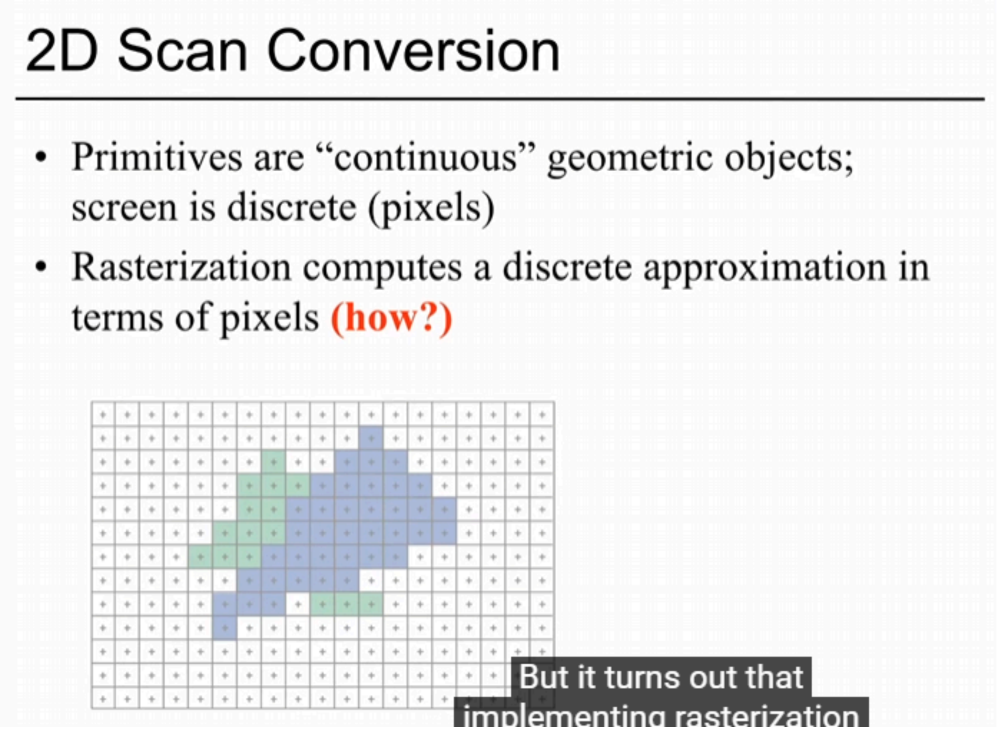
$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{far}-\text{near}} & -\frac{2*\text{far}*\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- z’=(az+b)/z =a+b/z
  - where a & b depend on near & far
- Similar enough to our basic idea:
  - z’=1/z

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- Rasterize triangle+ find which pixels shoud be lit

- 2D Scan Conversion

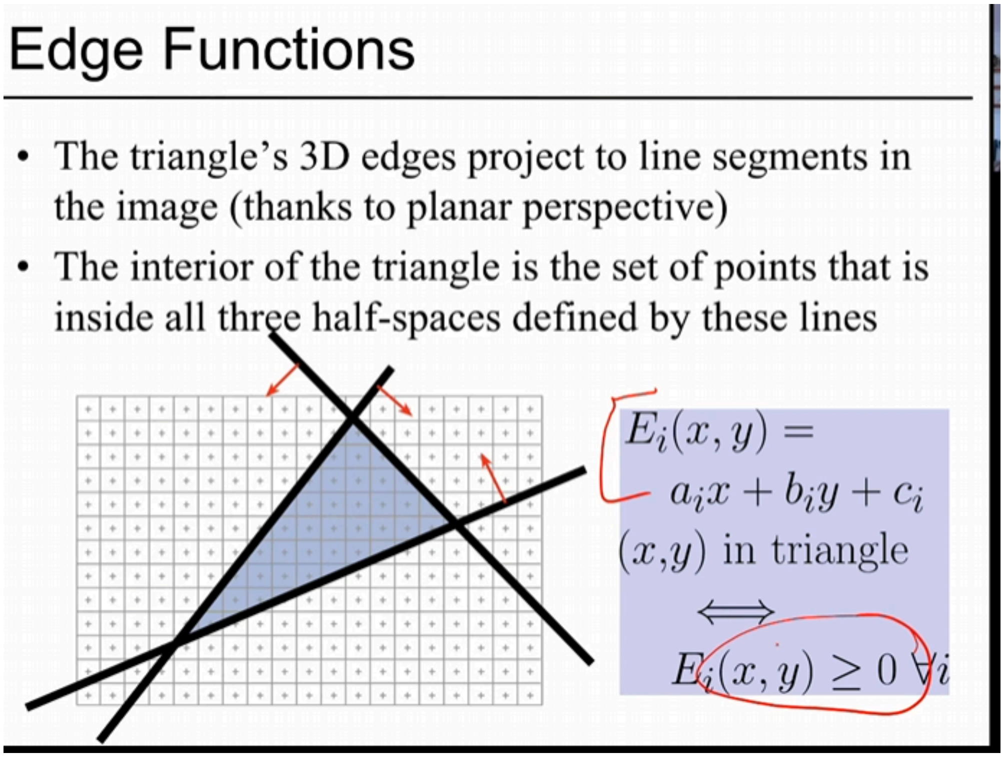


2D Scan Conversion

- Primitives are "continuous" geometric objects; screen is discrete (pixels)
- Rasterization computes a discrete approximation in terms of pixels **(how?)**

But it turns out that implementing rasterization

- Edge Functions



Edge Functions

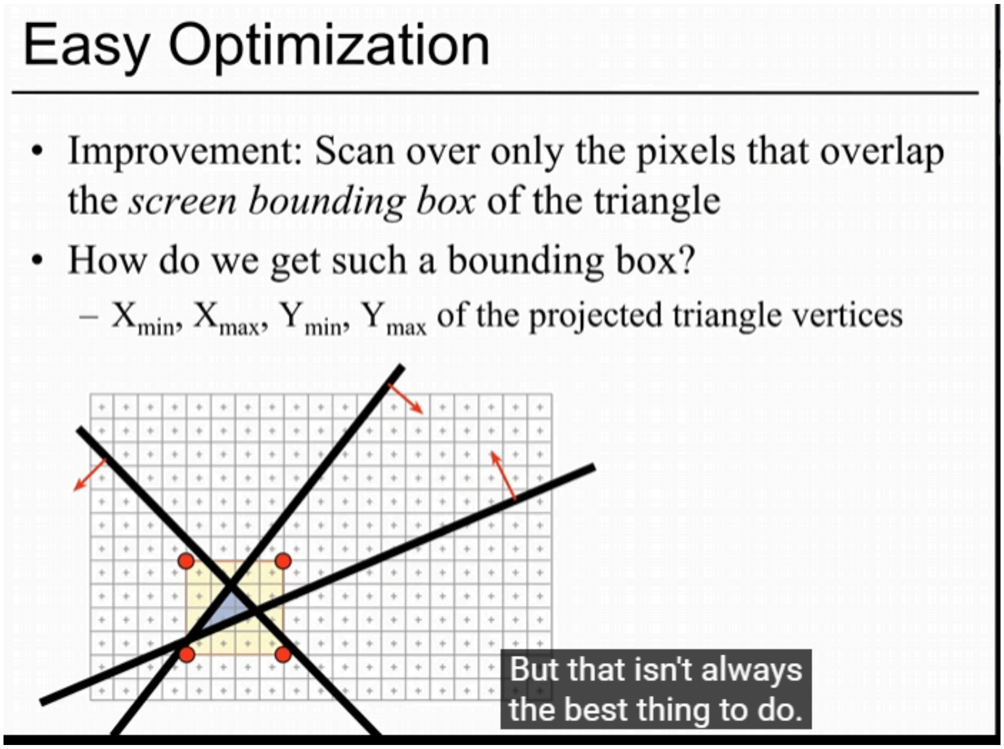
- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three half-spaces defined by these lines

$$E_i(x,y) = a_i x + b_i y + c_i$$

(x,y) in triangle

$$\iff E_i(x,y) \geq 0 \ \forall i$$

- Easy Optimization

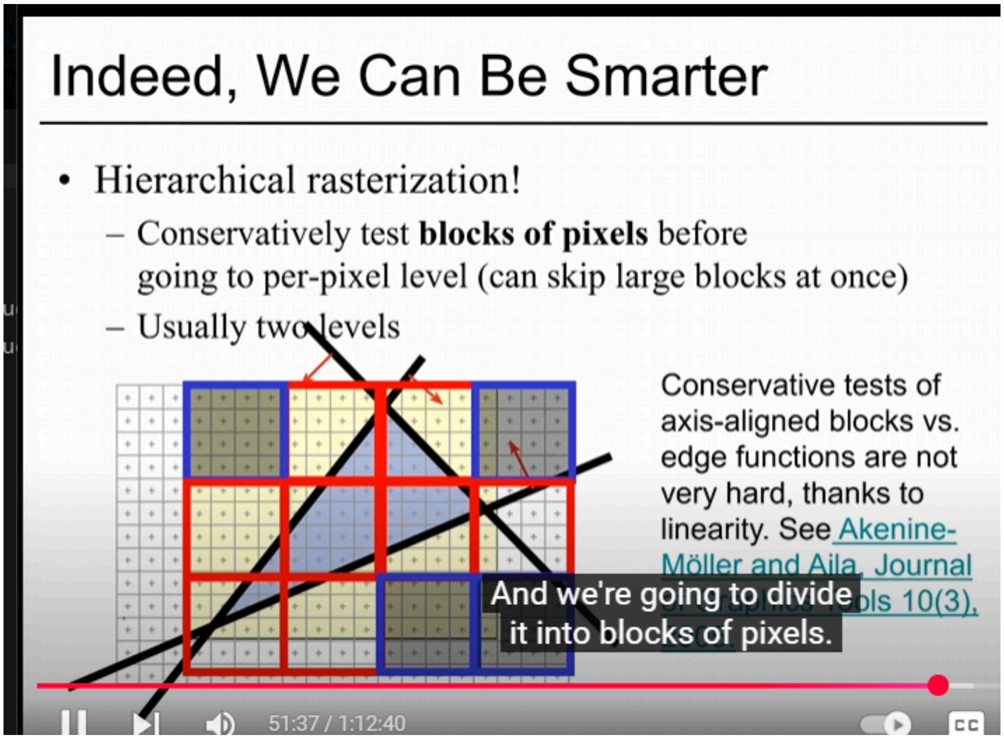


Easy Optimization

- Improvement: Scan over only the pixels that overlap the *screen bounding box* of the triangle
- How do we get such a bounding box?
  - X_{min}, X_{max}, Y_{min}, Y_{max} of the projected triangle vertices

But that isn't always the best thing to do.

- Hierarchical Rasterization



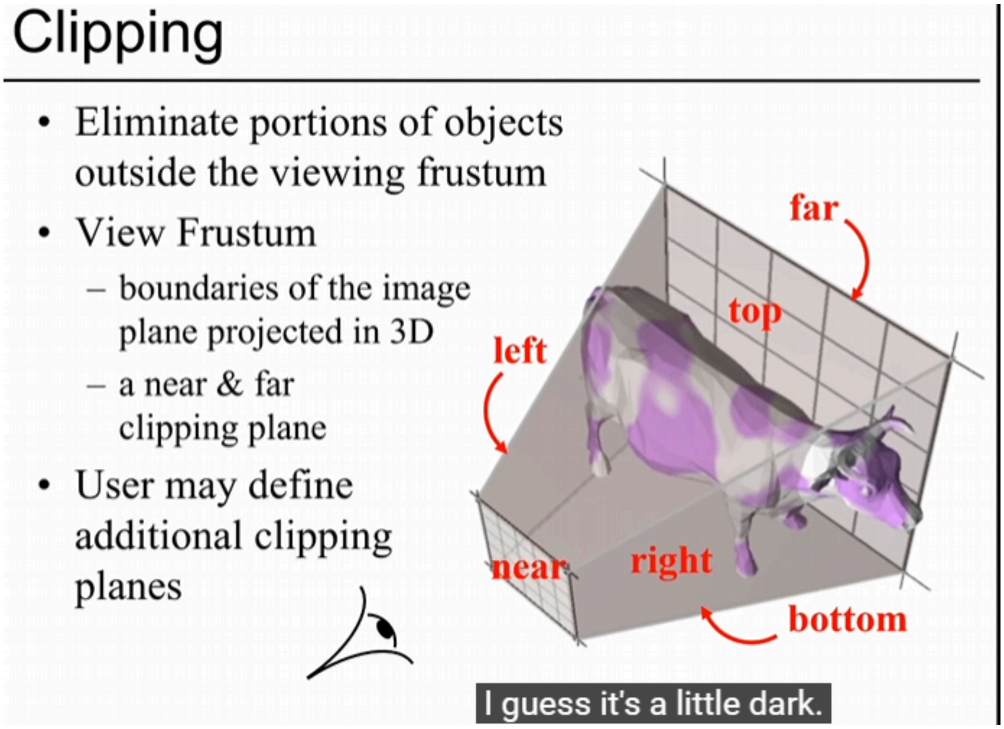
Indeed, We Can Be Smarter

- Hierarchical rasterization!
  - Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  - Usually two levels

Conservative tests of axis-aligned blocks vs. edge functions are not very hard, thanks to linearity. See Akenine-Möller and Aila, Journal of Graphics Tools 10(3), 2005.

And we're going to divide it into blocks of pixels.

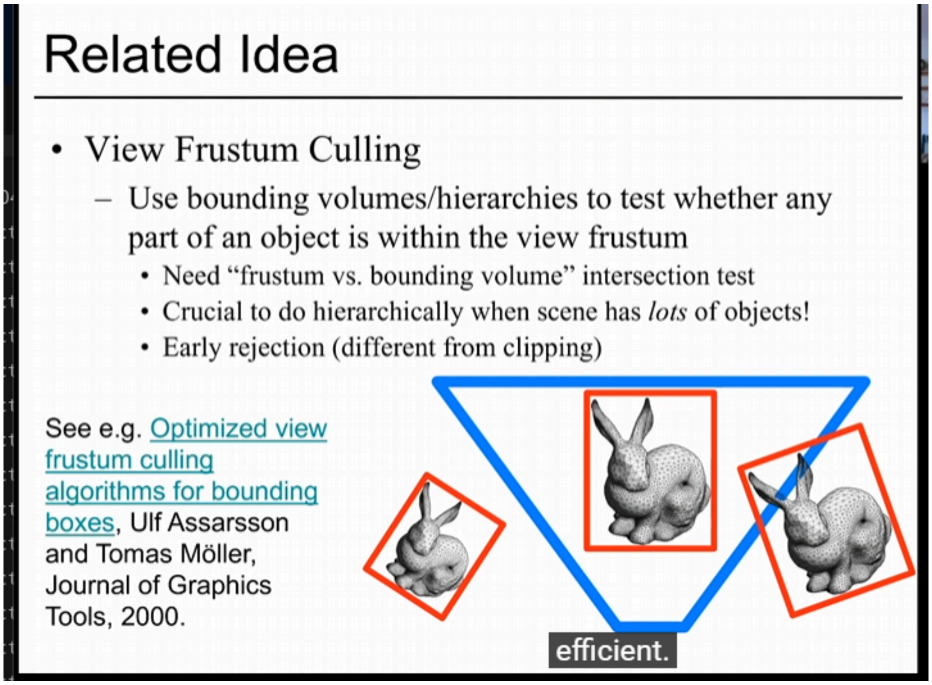
- Clipping



Clipping

- Eliminate portions of objects outside the viewing frustum
- View Frustum
  - boundaries of the image plane projected in 3D
  - a near & far clipping plane
- User may define additional clipping planes

  - Frustum Culling

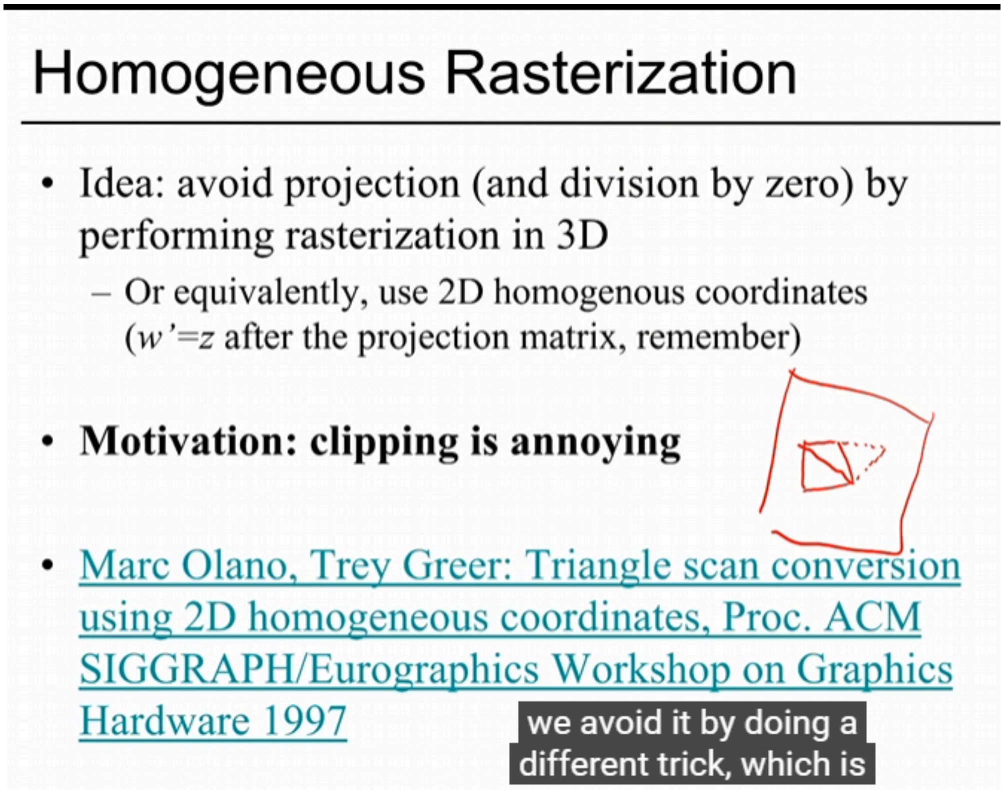


Related Idea

- View Frustum Culling
  - Use bounding volumes/hierarchies to test whether any part of an object is within the view frustum
    - Need "frustum vs. bounding volume" intersection test
    - Crucial to do hierarchically when scene has *lots* of objects!
    - Early rejection (different from clipping)

See e.g. Optimized view frustum culling algorithms for bounding boxes, Ulf Assarsson and Tomas Möller, Journal of Graphics Tools, 2000.

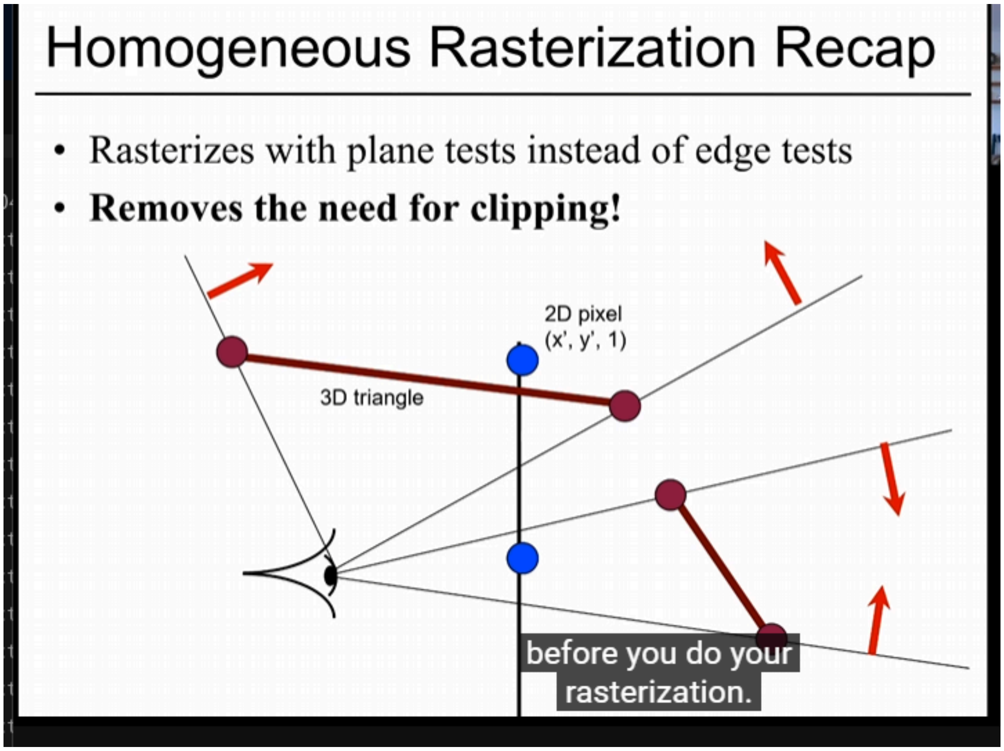
  - Homogeneous Rasterization



Homogeneous Rasterization

- Idea: avoid projection (and division by zero) by performing rasterization in 3D
  - Or equivalently, use 2D homogenous coordinates (*w'*=*z* after the projection matrix, remember)
- **Motivation: clipping is annoying**
- Marc Olano, Trey Greer: Triangle scan conversion using 2D homogeneous coordinates, Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware 1997

we avoid it by doing a different trick, which is



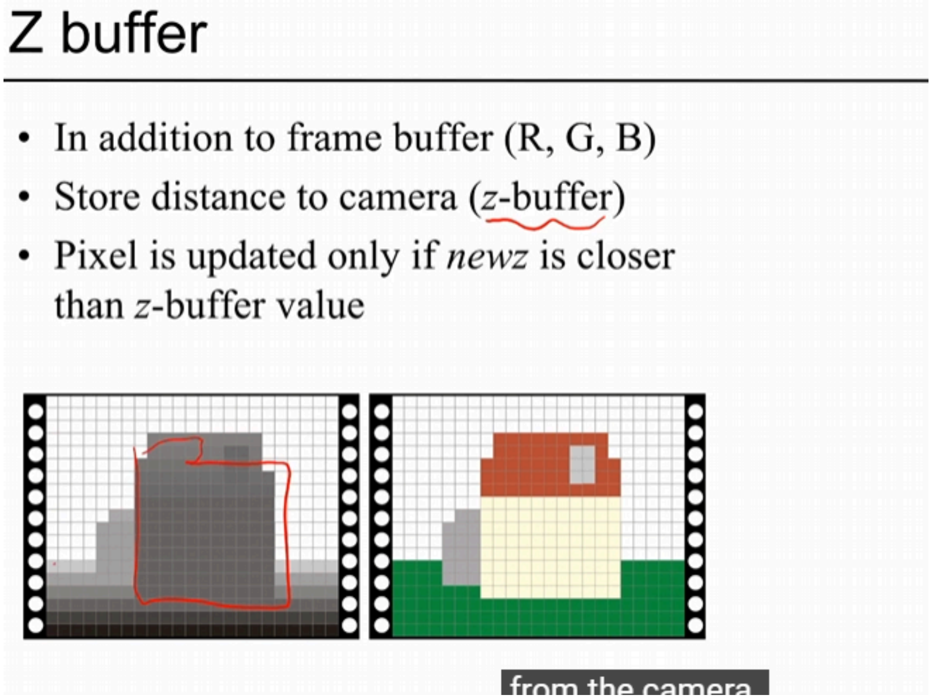
Homogeneous Rasterization Recap

- Rasterizes with plane tests instead of edge tests
- **Removes the need for clipping!**

before you do your rasterization.

- Compute Per Pixel Color
  - Pixel Shader
- Test visibility, update freame buffer
  - Painters algorithm
    - Draw 1 obj at a time
  - Z buffer

                    - distance to camera

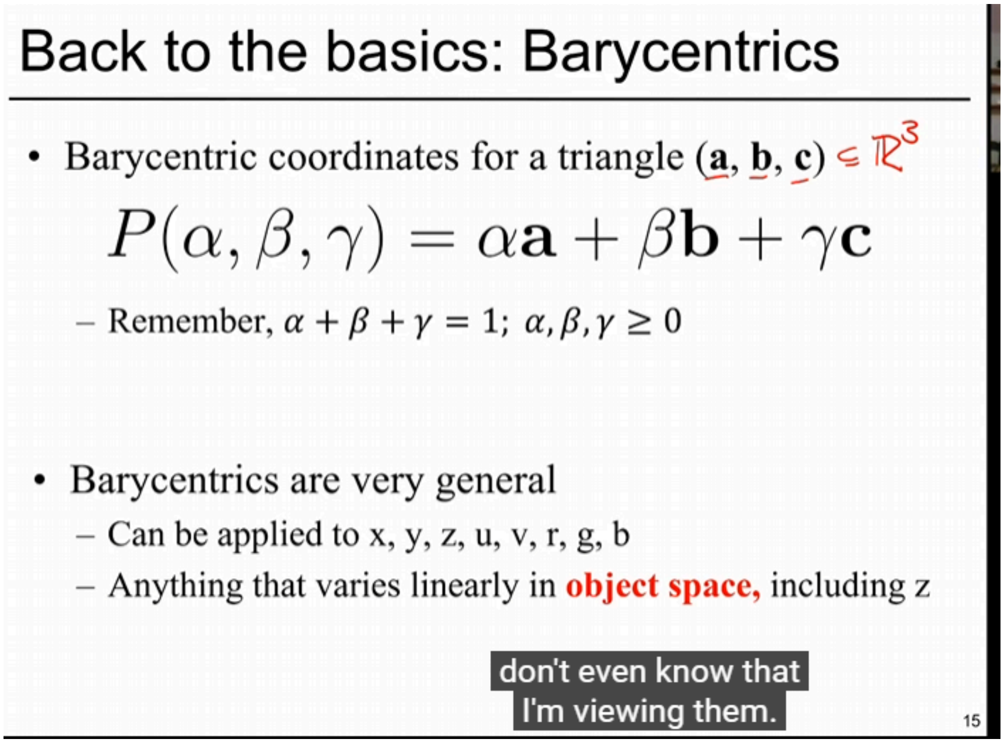


Z buffer

- In addition to frame buffer (R, G, B)
- Store distance to camera (*z*-buffer)
- Pixel is updated only if *newz* is closer than *z*-buffer value

from the camera

- **L18: Rasterization II: Z buffer, rasterized antialiasing**
    - Test visibility, update freame buffer (Continue of last lecture)
        - Interpolation in Screen Space![[Pasted image 20250121104158.png]
            - Find it depth by converted it back from 2D to 3D
                - Back to the basics: Barycentrics



Back to the basics: Barycentrics

- Barycentric coordinates for a triangle $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathbb{R}^3$

$$P(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

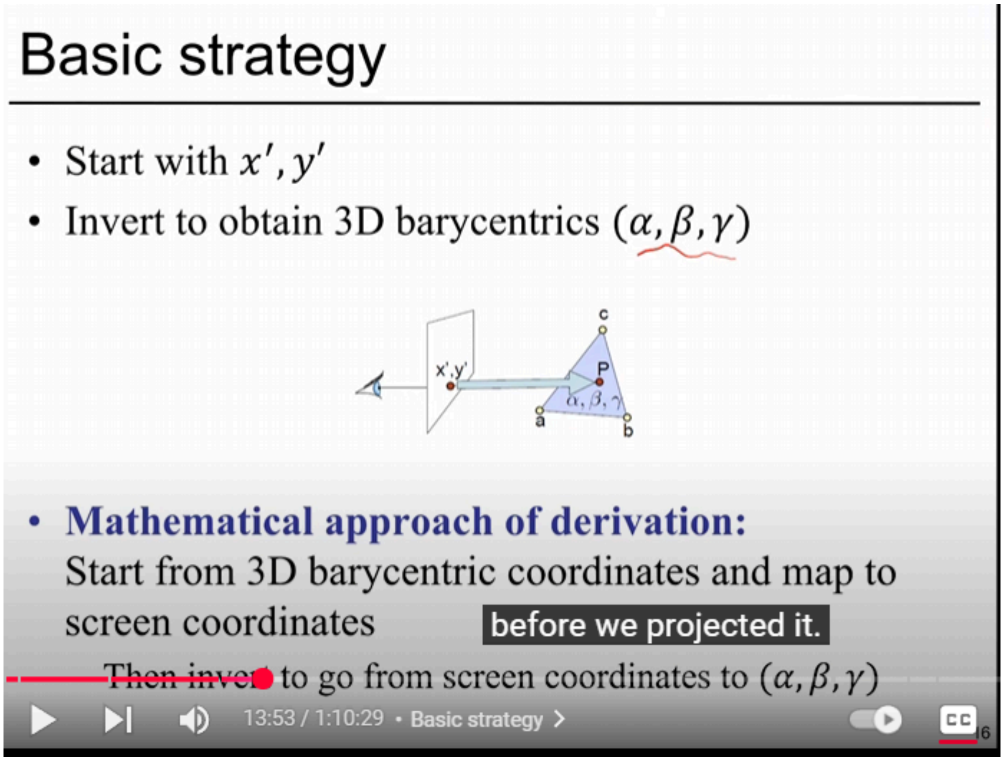
  – Remember, $\alpha + \beta + \gamma = 1;\ \alpha, \beta, \gamma \geq 0$

- Barycentrics are very general
  – Can be applied to x, y, z, u, v, r, g, b
  – Anything that varies linearly in **object space,** including z

don't even know that
I'm viewing them.

15

- Basic Strategy: get 3D barycentrics



Basic strategy

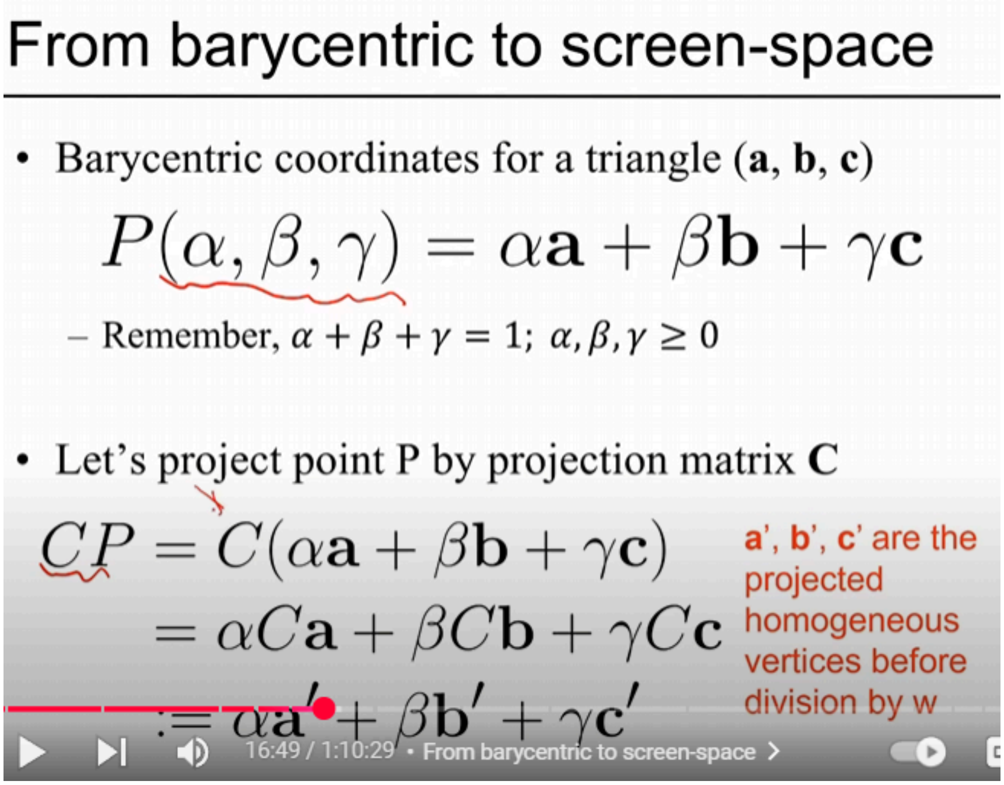
- Start with x', y'
- Invert to obtain 3D barycentrics $(\underbrace{\alpha, \beta, \gamma})$

- **Mathematical approach of derivation:**
  Start from 3D barycentric coordinates and map to screen coordinates

  before we projected it.

  Then invert to go from screen coordinates to (α, β, γ)

- From barycentric to screen-space (before homogenization)



From barycentric to screen-space

- Barycentric coordinates for a triangle (**a**, **b**, **c**)

$$\underbrace{P(\alpha, \beta, \gamma)} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$$

  – Remember, $\alpha + \beta + \gamma = 1;\ \alpha, \beta, \gamma \geq 0$

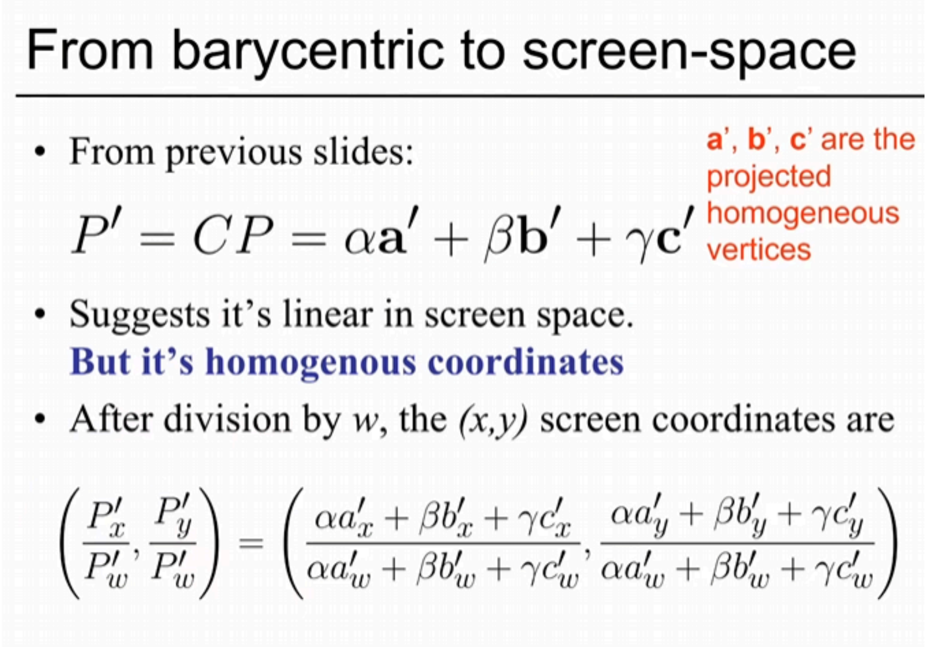
- Let's project point P by projection matrix **C**

$$\begin{aligned} \underbrace{CP} &= C(\alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}) \\ &= \alpha C\mathbf{a} + \beta C\mathbf{b} + \gamma C\mathbf{c} \\ &:= \alpha\mathbf{a}' + \beta\mathbf{b}' + \gamma\mathbf{c}' \end{aligned}$$

a', **b**', **c**' are the projected homogeneous vertices before division by w

  - CP is projection on 2D of the 3D triangle

- Dehomongenized point on the computer screen



From barycentric to screen-space

- From previous slides:

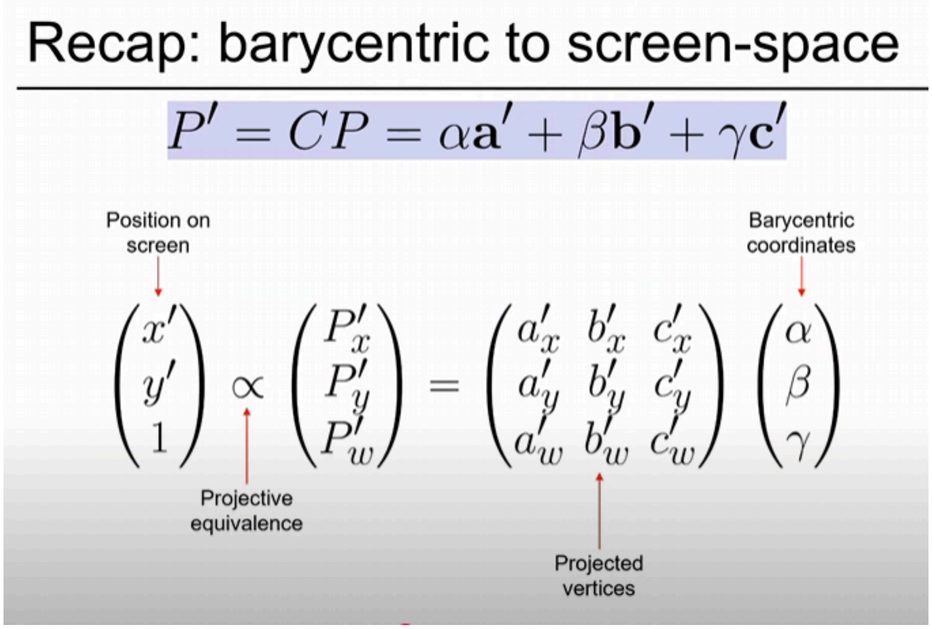
$$P' = CP = \alpha\mathbf{a}' + \beta\mathbf{b}' + \gamma\mathbf{c}'$$

a', **b**', **c**' are the projected homogeneous vertices

- Suggests it's linear in screen space.
  But it's homogenous coordinates
- After division by *w*, the *(x,y)* screen coordinates are

$$\left(\frac{P'_x}{P'_w}, \frac{P'_y}{P'_w}\right) = \left(\frac{\alpha a'_x + \beta b'_x + \gamma c'_x}{\alpha a'_w + \beta b'_w + \gamma c'_w}, \frac{\alpha a'_y + \beta b'_y + \gamma c'_y}{\alpha a'_w + \beta b'_w + \gamma c'_w}\right)$$

- Goal: calculate Barycentric coordinates in 3D



Recap: barycentric to screen-space

$$P' = CP = \alpha\mathbf{a}' + \beta\mathbf{b}' + \gamma\mathbf{c}'$$

Position on screen

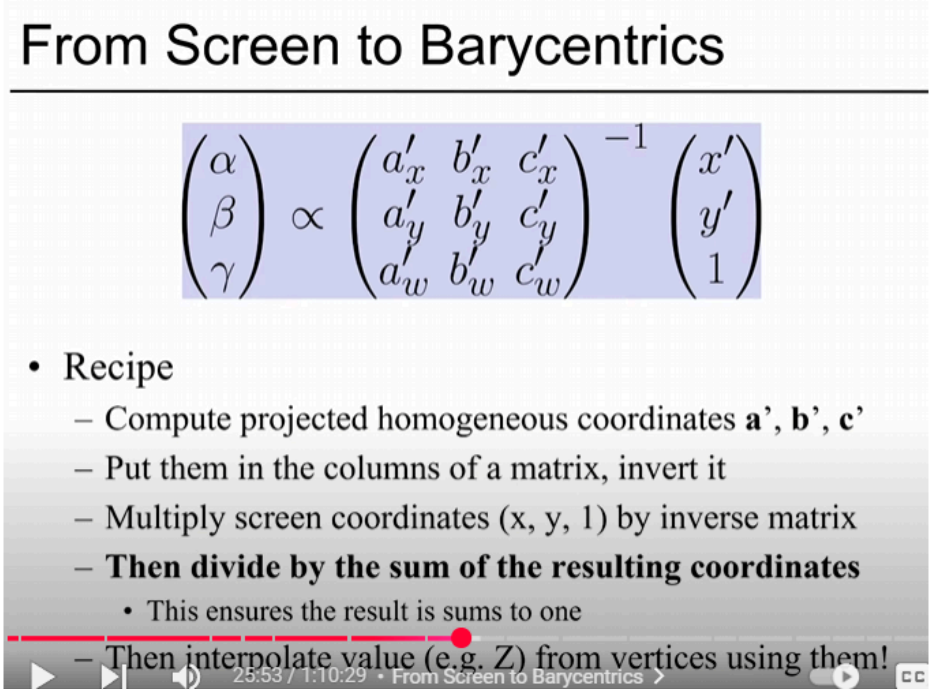
Barycentric coordinates

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \propto \begin{pmatrix} P'_x \\ P'_y \\ P'_w \end{pmatrix} = \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_w & b'_w & c'_w \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$$

Projective equivalence

Projected vertices

- How to Calculate a b r

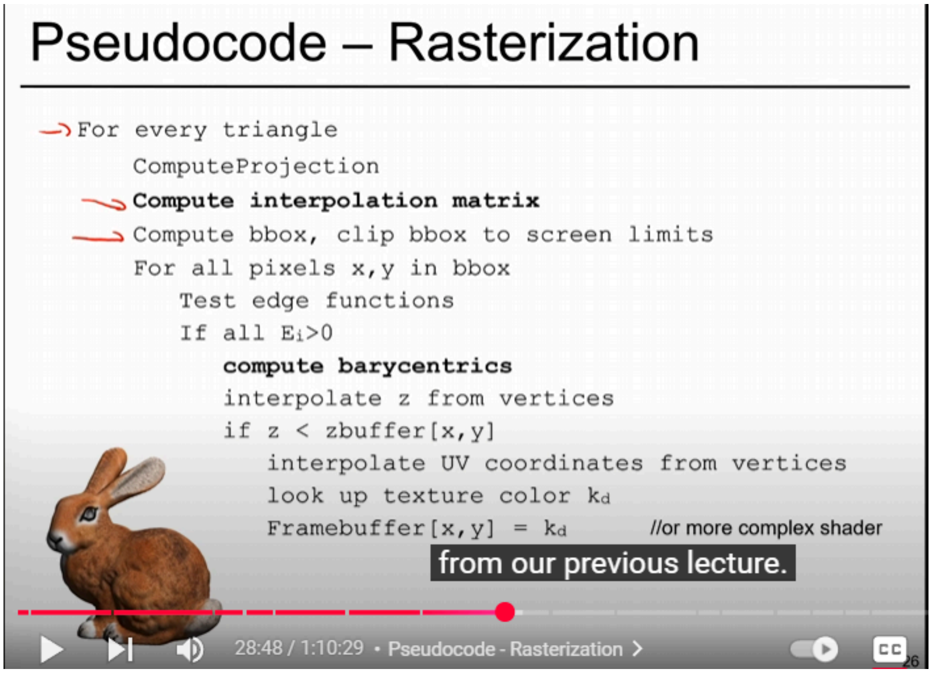


From Screen to Barycentrics

$$\begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} \propto \begin{pmatrix}a'_x & b'_x & c'_x\\ a'_y & b'_y & c'_y\\ a'_w & b'_w & c'_w\end{pmatrix}^{-1} \begin{pmatrix}x'\\ y'\\ 1\end{pmatrix}$$

- Recipe
  - Compute projected homogeneous coordinates **a'**, **b'**, **c'**
  - Put them in the columns of a matrix, invert it
  - Multiply screen coordinates (x, y, 1) by inverse matrix
  - **Then divide by the sum of the resulting coordinates**
    - This ensures the result is sums to one
  - Then interpolate value (e.g. Z) from vertices using them!

- Pseudocode - Rasterization



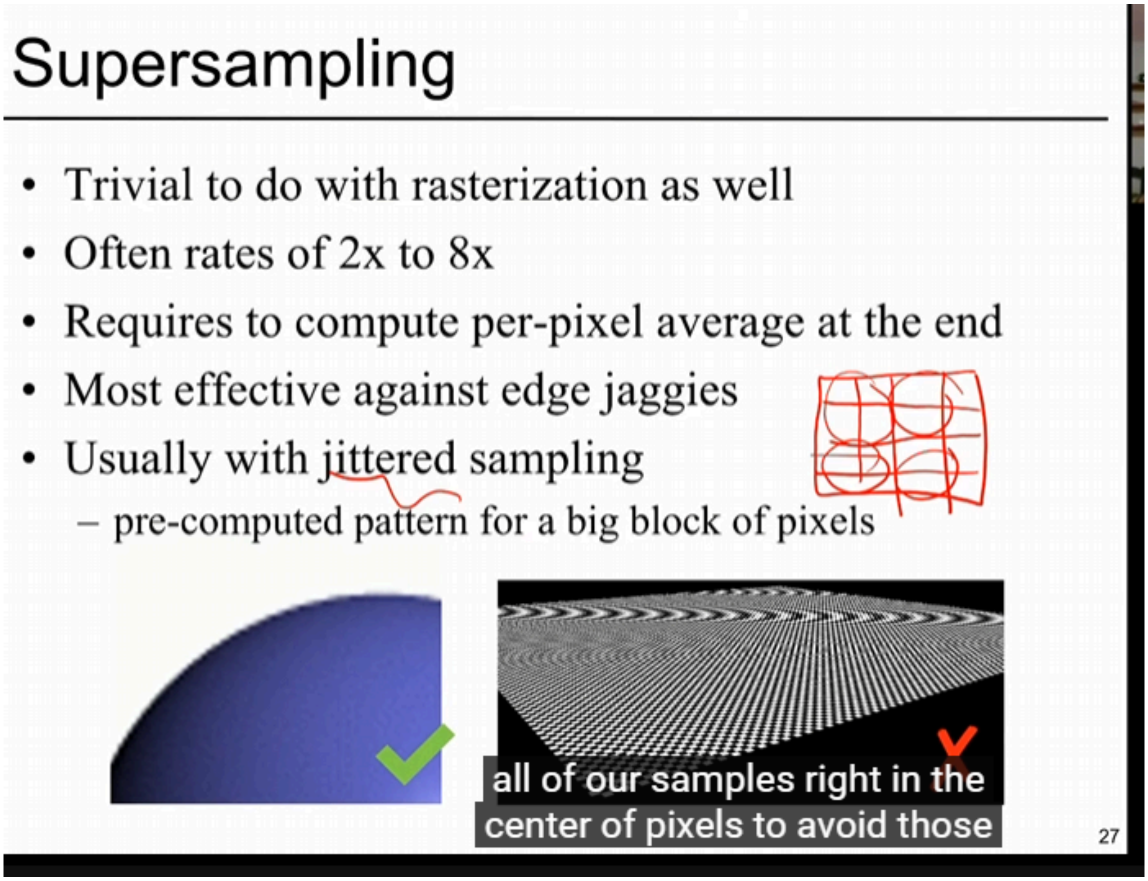
Pseudocode – Rasterization

```
For every triangle
    ComputeProjection
    Compute interpolation matrix
    Compute bbox, clip bbox to screen limits
    For all pixels x,y in bbox
        Test edge functions
        If all $E_i$>0
            compute barycentrics
            interpolate z from vertices
            if z < zbuffer[x,y]
                interpolate UV coordinates from vertices
                look up texture color $k_d$
                Framebuffer[x,y] = $k_d$        //or more complex shader
```

from our previous lecture.
28:48 / 1:10:29 • Pseudocode - Rasterization
CC
26

- Rasterization Anti-aliasing

- Supersampling



Supersampling

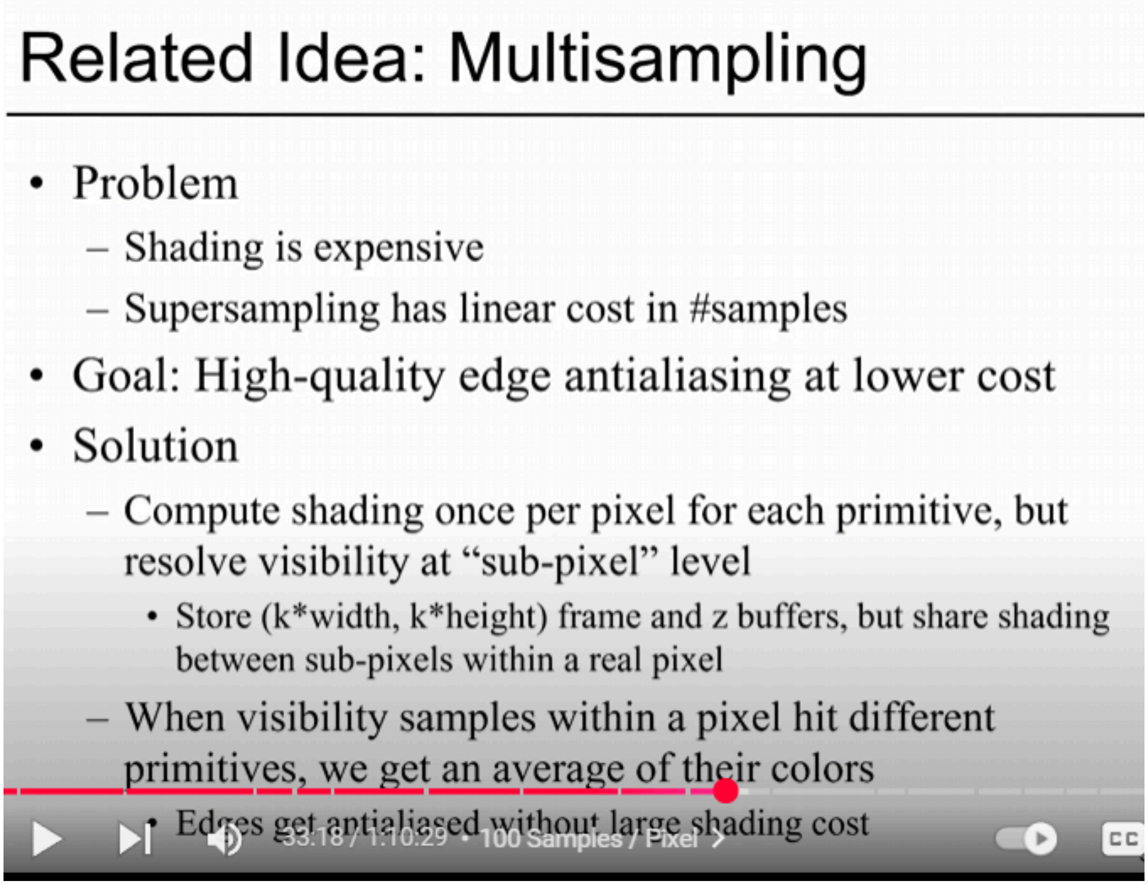
- Trivial to do with rasterization as well
- Often rates of 2x to 8x
- Requires to compute per-pixel average at the end
- Most effective against edge jaggies
- Usually with jittered sampling
    - pre-computed pattern for a big block of pixels

all of our samples right in the center of pixels to avoid those

27

    - Render more than 1 sample per pixel, average the result
        - Scale up the the image, average it

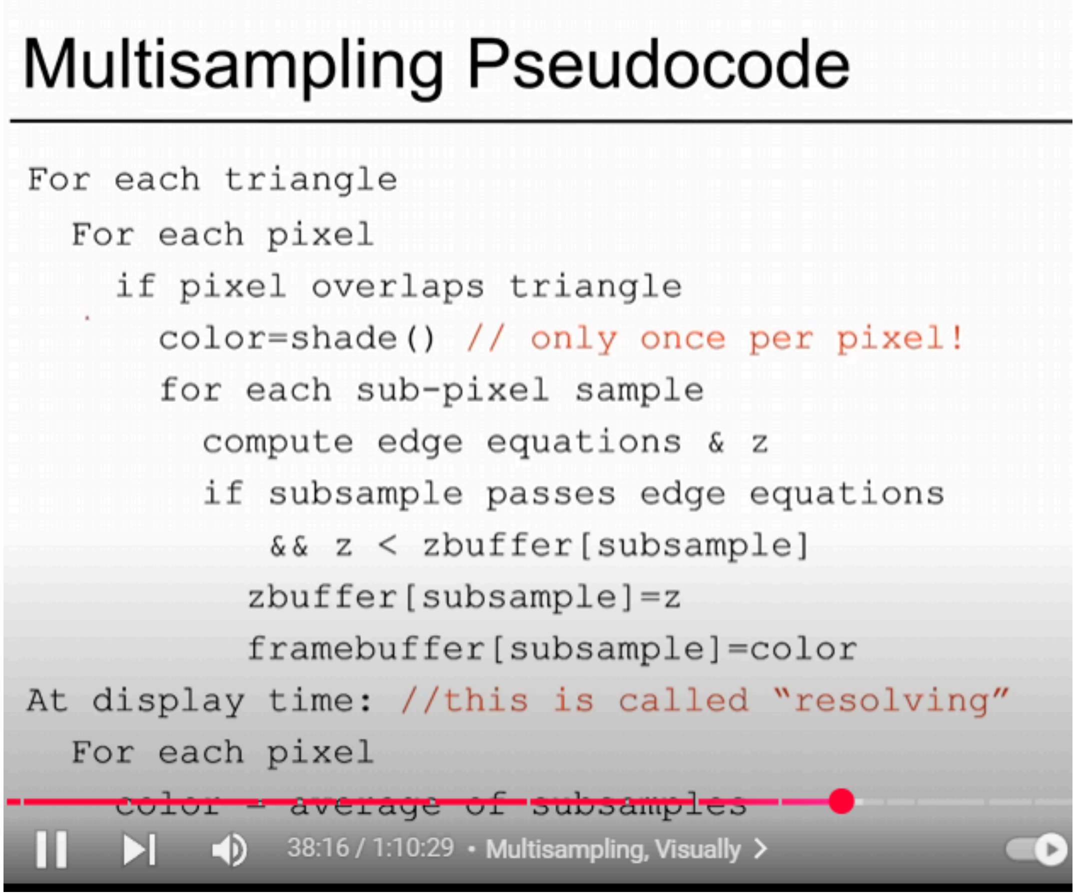
- Multisampling



Related Idea: Multisampling

- Problem
  - Shading is expensive
  - Supersampling has linear cost in #samples
- Goal: High-quality edge antialiasing at lower cost
- Solution
  - Compute shading once per pixel for each primitive, but resolve visibility at “sub-pixel” level
    - Store (k*width, k*height) frame and z buffers, but share shading between sub-pixels within a real pixel
  - When visibility samples within a pixel hit different primitives, we get an average of their colors
    - Edges get antialiased without large shading cost

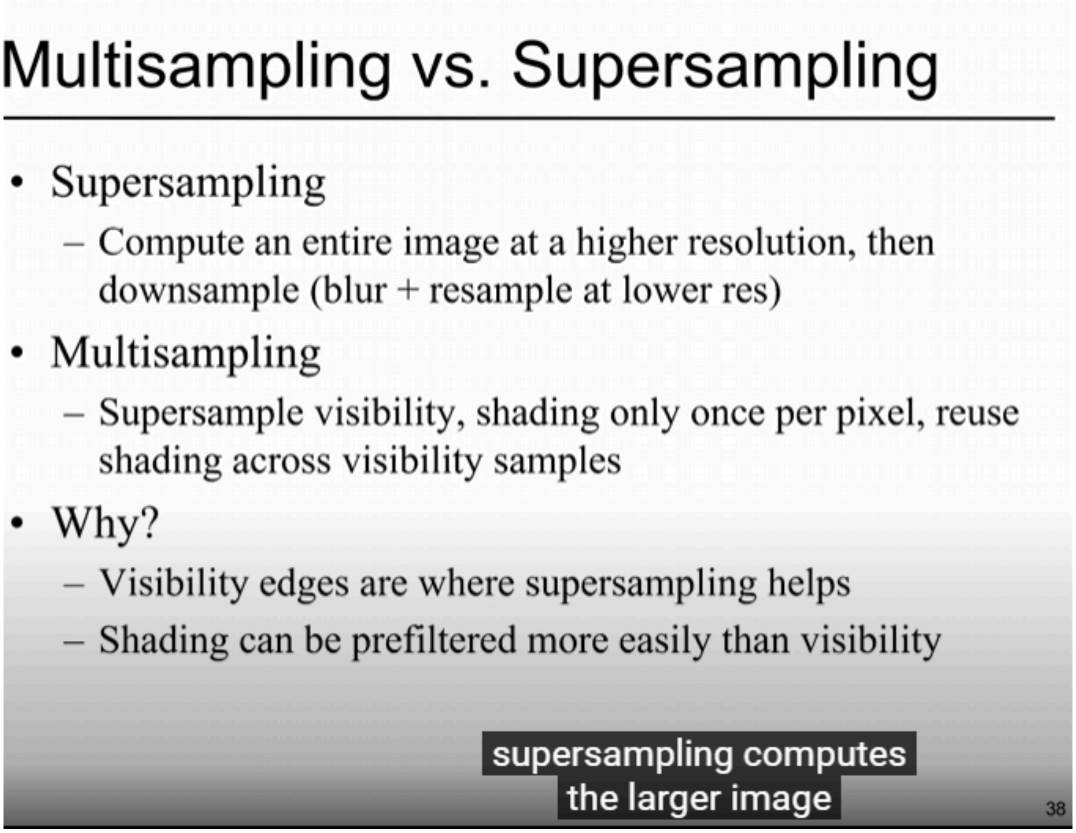
  - average the color of the pixel which has multiple triangle
  - Multisampling Pseudocode



Multisampling Pseudocode

```
For each triangle
  For each pixel
    if pixel overlaps triangle
      color=shade() // only once per pixel!
      for each sub-pixel sample
        compute edge equations & z
        if subsample passes edge equations
           && z < zbuffer[subsample]
          zbuffer[subsample]=z
          framebuffer[subsample]=color
At display time: //this is called “resolving”
  For each pixel
    color = average of subsamples
```

- Comparision

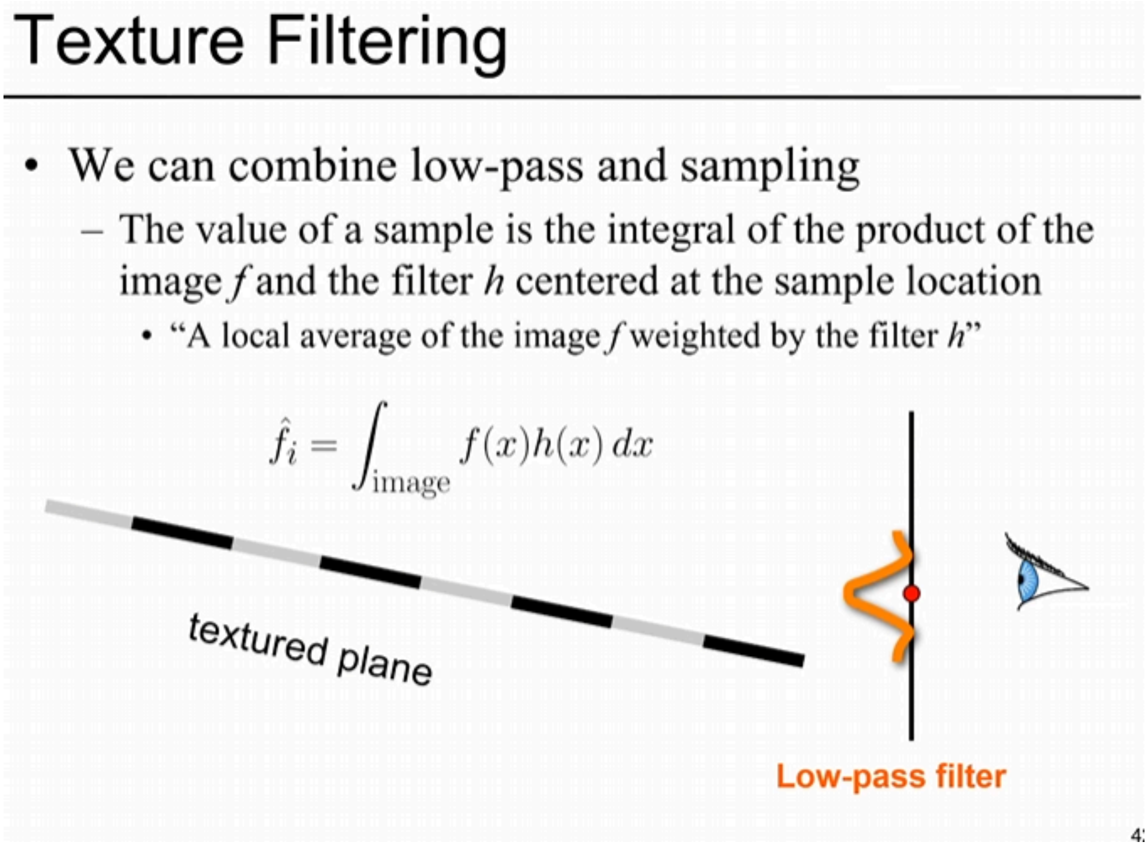


Multisampling vs. Supersampling

- Supersampling
  - Compute an entire image at a higher resolution, then downsample (blur + resample at lower res)
- Multisampling
  - Supersample visibility, shading only once per pixel, reuse shading across visibility samples
- Why?
  - Visibility edges are where supersampling helps
  - Shading can be prefiltered more easily than visibility

supersampling computes
the larger image

- Texture Filtering



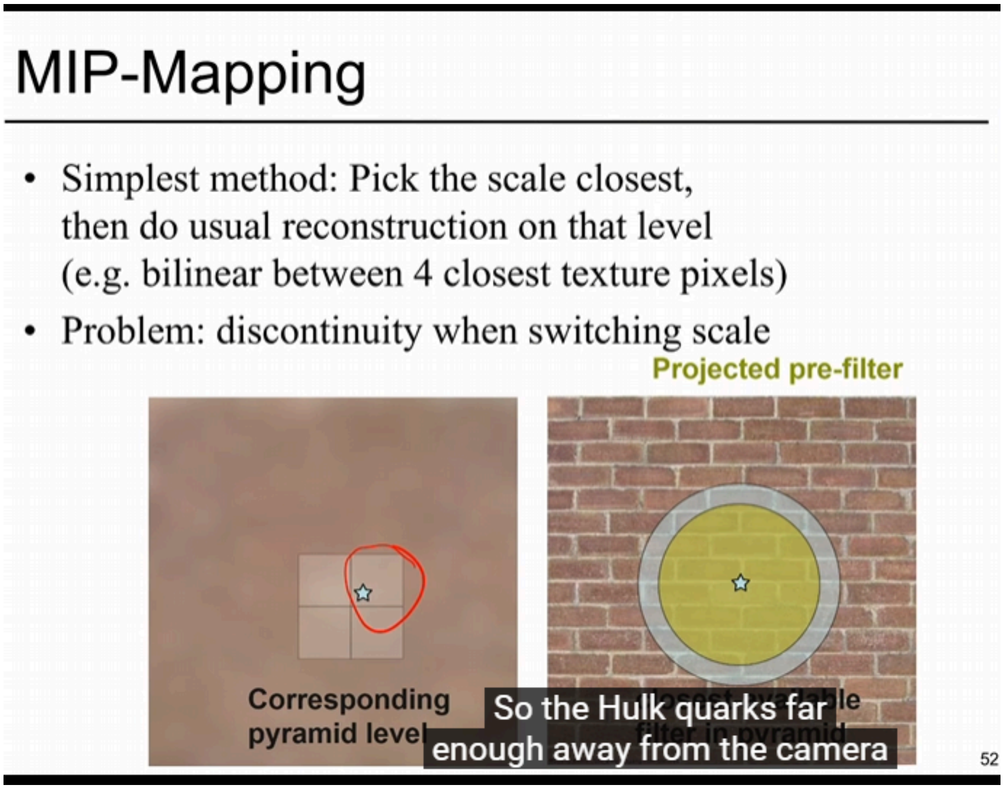
Texture Filtering

- We can combine low-pass and sampling
  - The value of a sample is the integral of the product of the image *f* and the filter *h* centered at the sample location
    - "A local average of the image *f* weighted by the filter *h*"

$$\hat{f}_i = \int_{\text{image}} f(x)h(x)\,dx$$

- Prefiltering
  - Apply Low-pass filter to the texture to blur it

- MIP-Mapping



MIP-Mapping

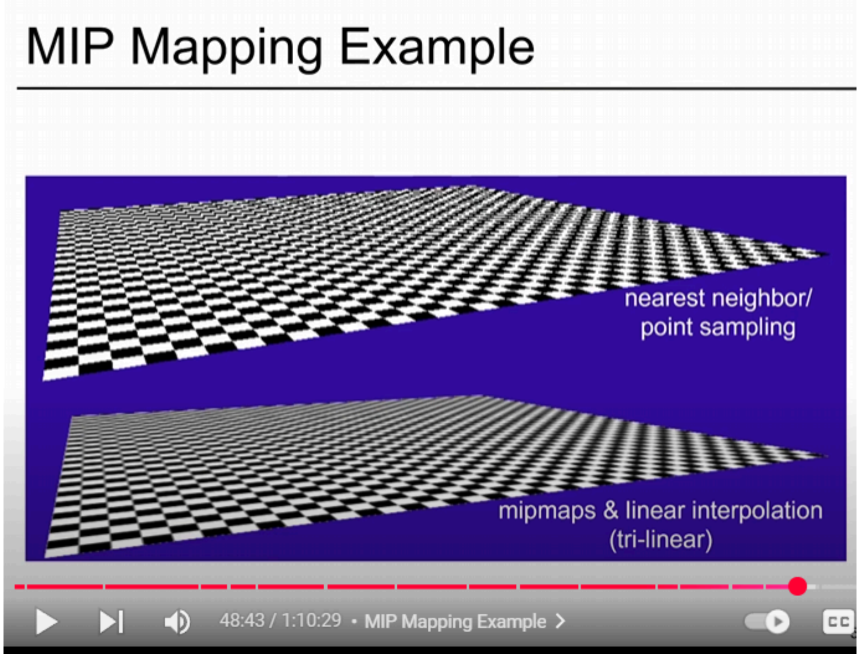
- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)
- Problem: discontinuity when switching scale

Projected pre-filter
Corresponding pyramid level
Nearest available filter in pyramid

So the Hulk quarks far enough away from the camera

52

  - Tri-Linear MIP-Mapping
    - Use two closet scales, compute reconstruction results from both, and linearly interpolate between them
    - Example

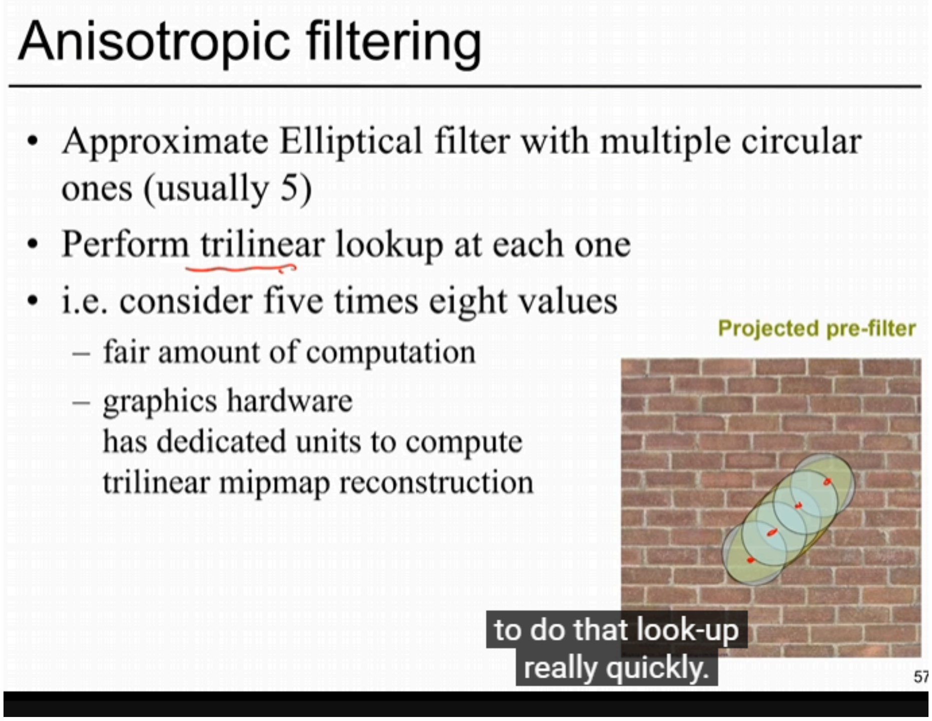


MIP Mapping Example

nearest neighbor/ point sampling
mipmaps & linear interpolation (tri-linear)
48:43 / 1:10:29 • MIP Mapping Example ›

  - MIP Maps only store 1/3 more space

- Anisotropic filthering



Anisotropic filtering

- Approximate Elliptical filter with multiple circular ones (usually 5)
- Perform trilinear lookup at each one
- i.e. consider five times eight values
  - fair amount of computation
  - graphics hardware has dedicated units to compute trilinear mipmap reconstruction

to do that look-up really quickly.

57

- Comparison

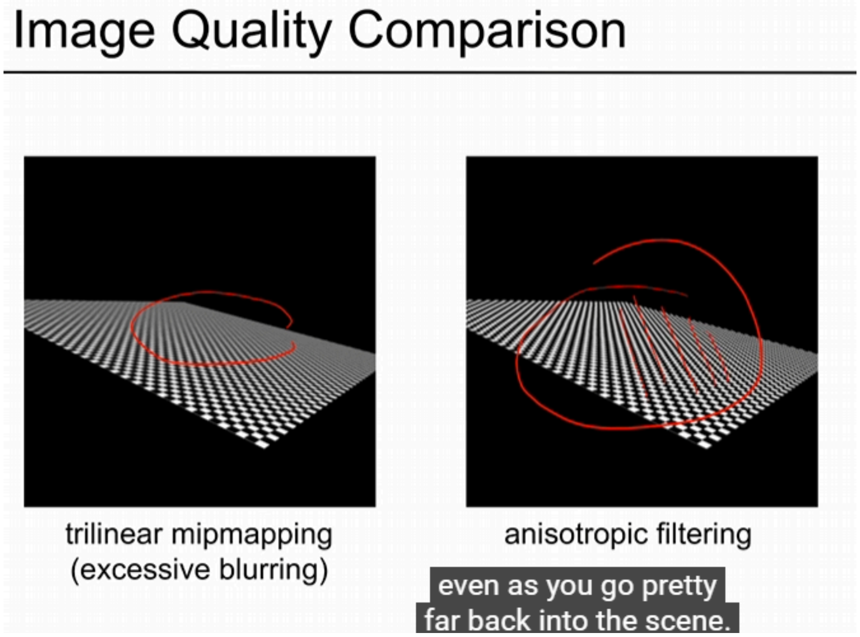


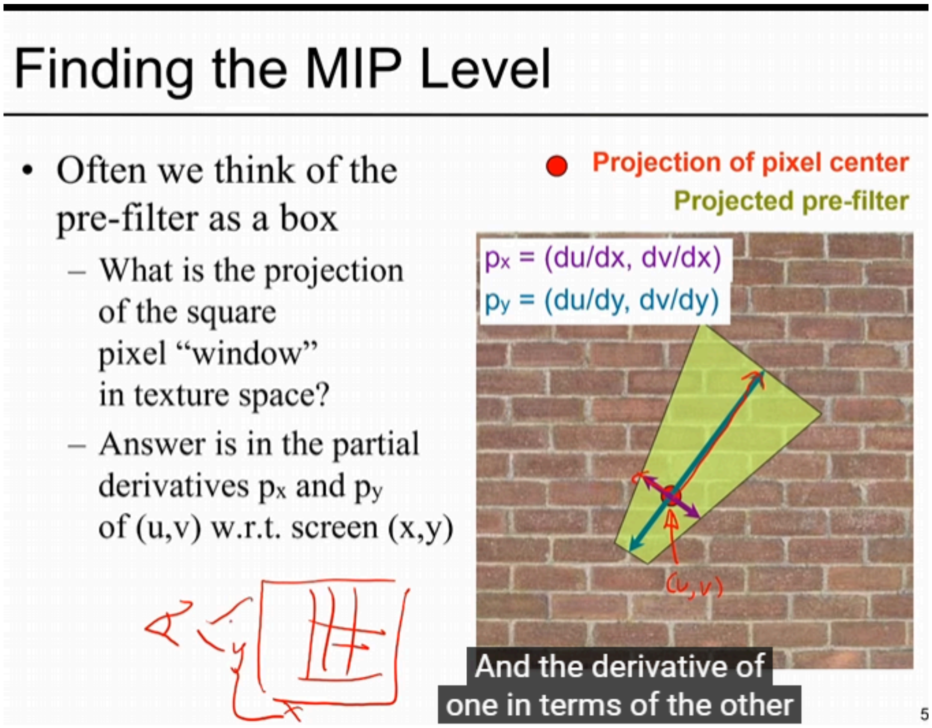
Image Quality Comparison

trilinear mipmapping (excessive blurring)

anisotropic filtering

even as you go pretty far back into the scene.

- Finding the MIP level



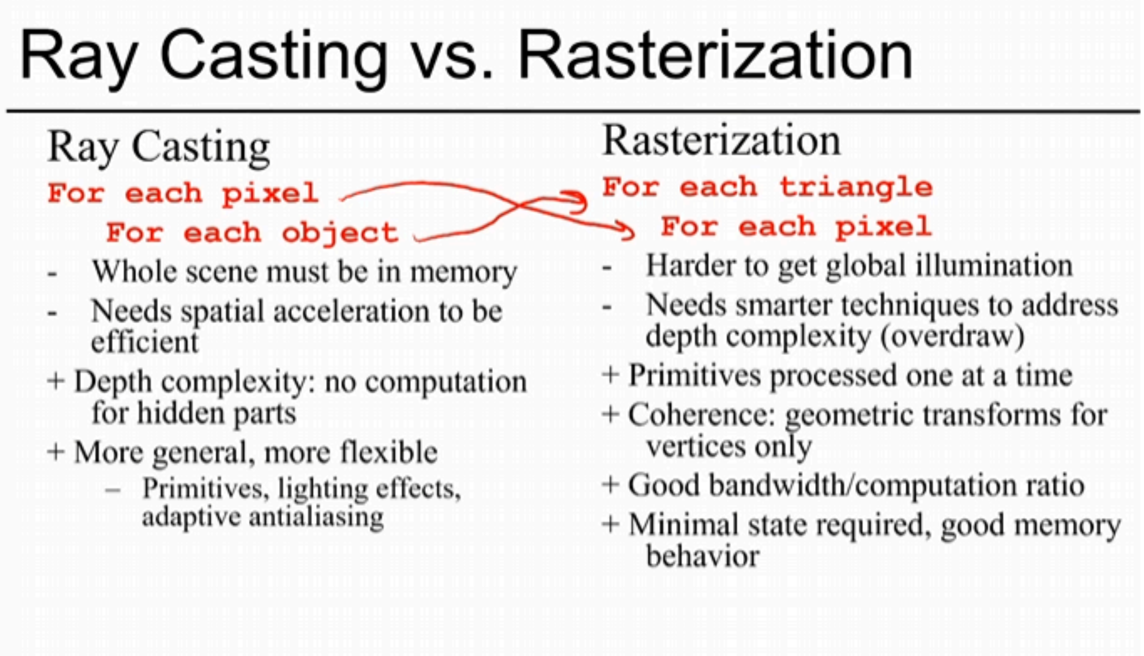
Finding the MIP Level

- Often we think of the pre-filter as a box
  - What is the projection of the square pixel "window" in texture space?
  - Answer is in the partial derivatives p_x and p_y of (u,v) w.r.t. screen (x,y)

And the derivative of one in terms of the other

5

- Review
  - Ray Casting vs. Rasterization



Ray Casting vs. Rasterization

Ray Casting

```
For each pixel
    For each object
```

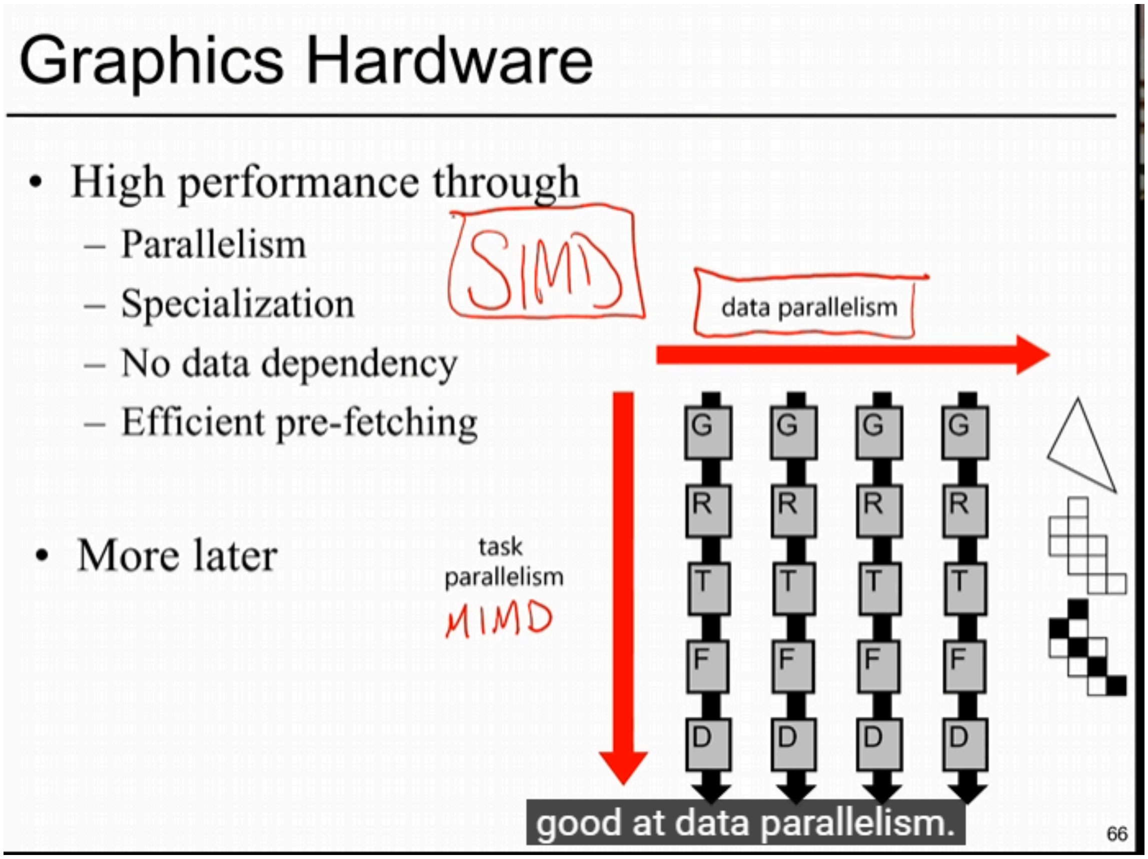
- - Whole scene must be in memory
- - Needs spatial acceleration to be efficient
- + Depth complexity: no computation for hidden parts
- + More general, more flexible
  - Primitives, lighting effects, adaptive antialiasing

Rasterization

```
For each triangle
    For each pixel
```

- - Harder to get global illumination
- - Needs smarter techniques to address depth complexity (overdraw)
- + Primitives processed one at a time
- + Coherence: geometric transforms for vertices only
- + Good bandwidth/computation ratio
- + Minimal state required, good memory behavior

    - Graphics Hardware



Graphics Hardware

- High performance through
  - Parallelism
  - Specialization
  - No data dependency
  - Efficient pre-fetching
- More later

SIMD

data parallelism

task parallelism

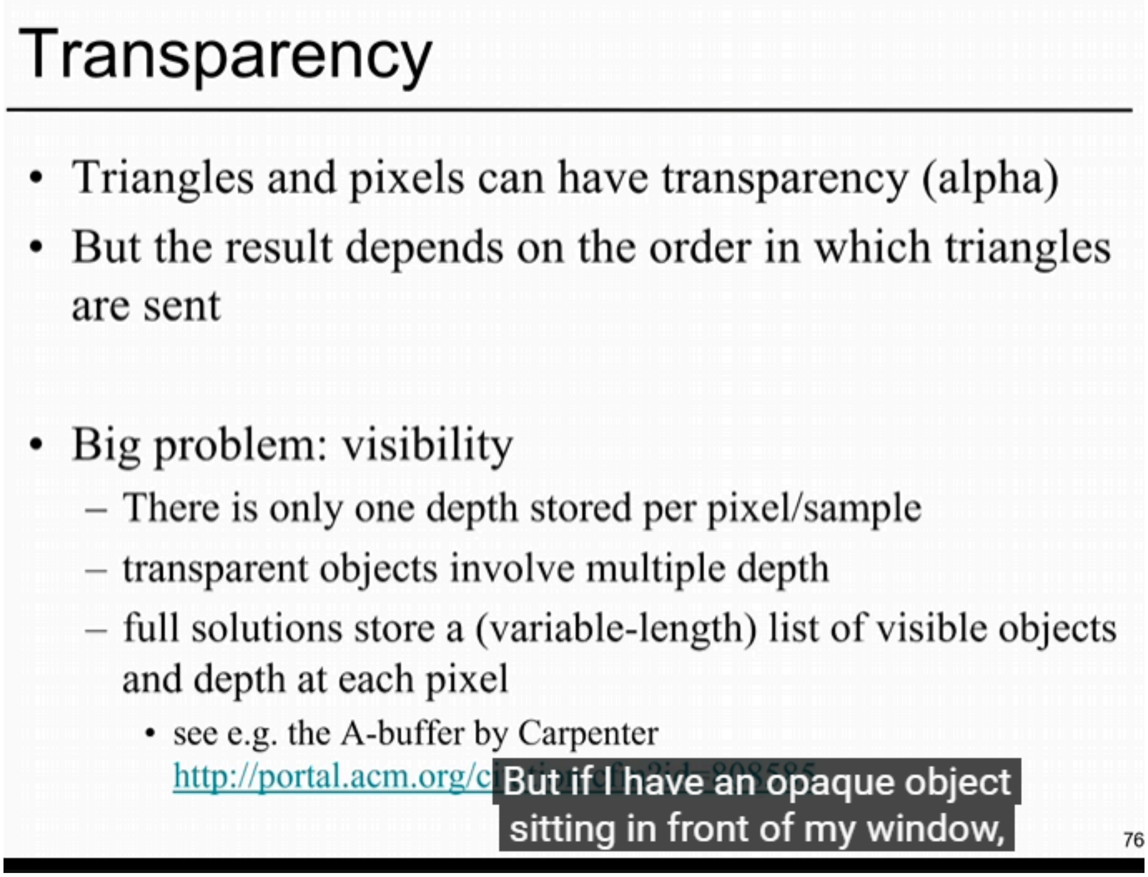
MIMD

good at data parallelism.

66

    - Movies
        - Combination
    - Games (2020)
        - Mostly Rasterization
        - Some Ray Tracing
    - CAD-CMD
        - Ray Tracing
    - Architecture
        - Ray Tracing
    - Vitual Reality
        - Rasterization
    - Visualization
        - Combination
    - Medical Imaging
        - Combination
- Challenges of Rasterization

- Transparency



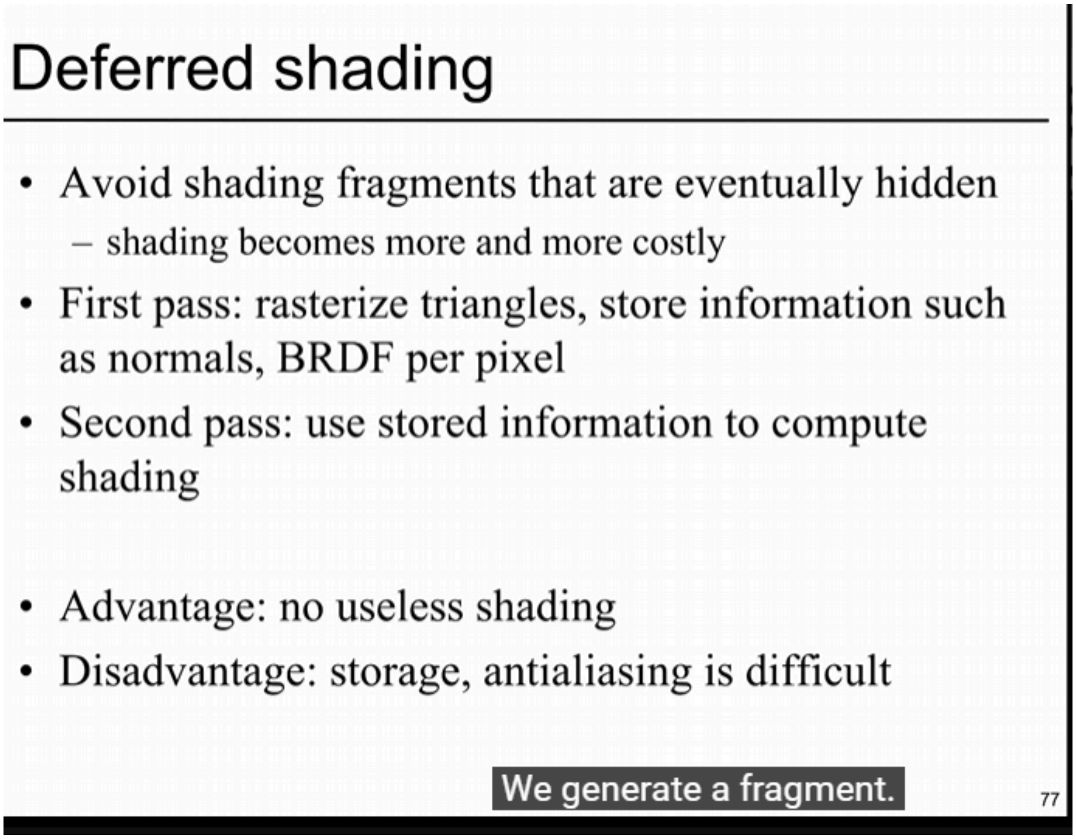
Transparency

- Triangles and pixels can have transparency (alpha)
- But the result depends on the order in which triangles are sent
- Big problem: visibility
  - There is only one depth stored per pixel/sample
  - transparent objects involve multiple depth
  - full solutions store a (variable-length) list of visible objects and depth at each pixel
    - see e.g. the A-buffer by Carpenter http://portal.acm.org/ci

But if I have an opaque object sitting in front of my window,

76

- Alternative approaches
  - Reyes (Pixar's Renderman)
  - Defered shading



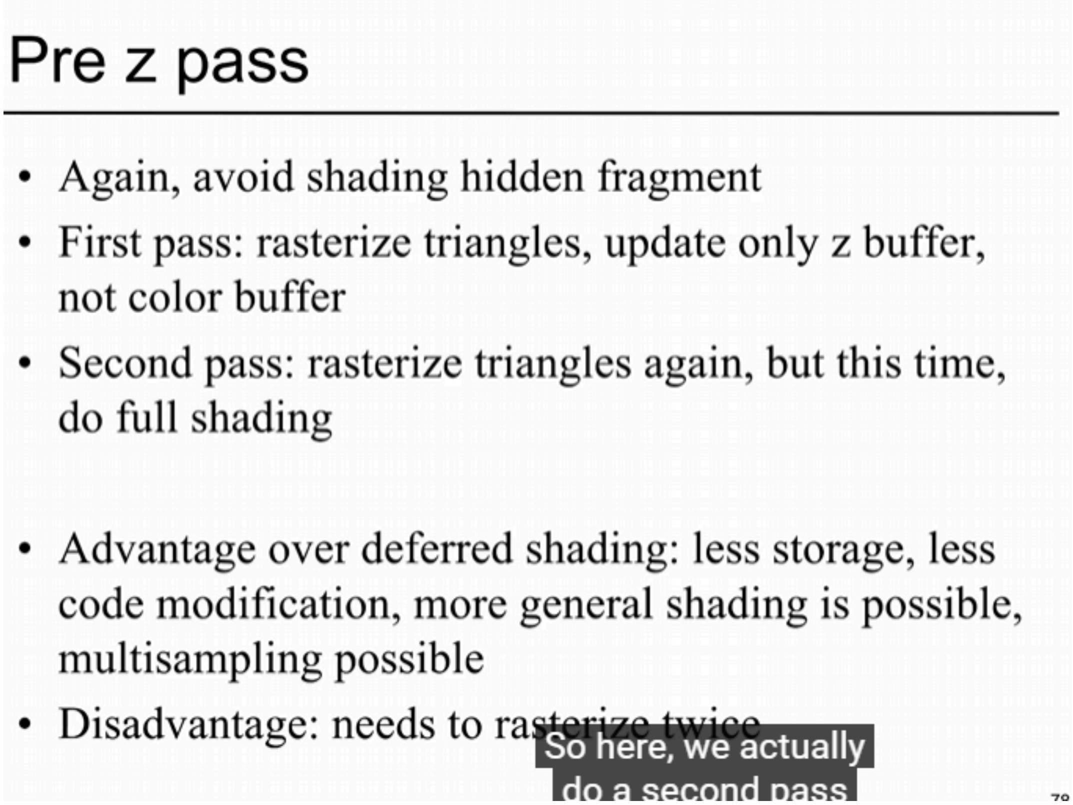
Deferred shading

- Avoid shading fragments that are eventually hidden
  - shading becomes more and more costly
- First pass: rasterize triangles, store information such as normals, BRDF per pixel
- Second pass: use stored information to compute shading
- Advantage: no useless shading
- Disadvantage: storage, antialiasing is difficult

We generate a fragment.

77

    - Pre-Z pass



Pre z pass

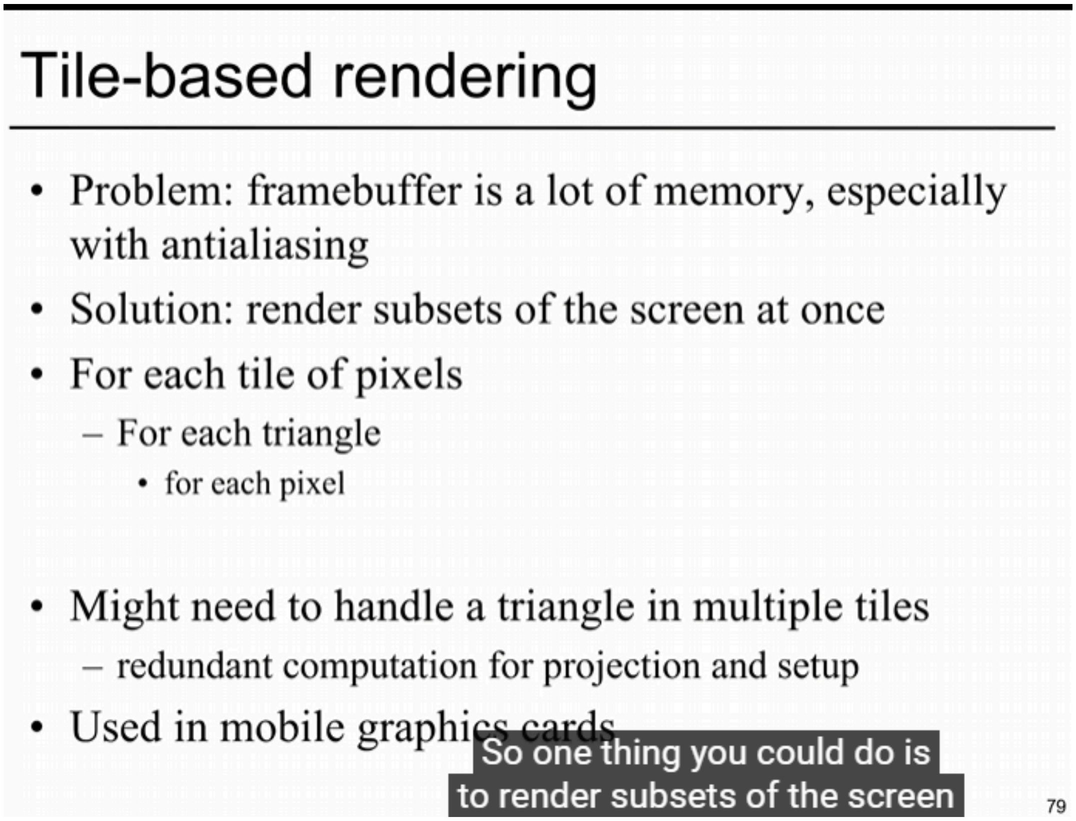
- Again, avoid shading hidden fragment
- First pass: rasterize triangles, update only z buffer, not color buffer
- Second pass: rasterize triangles again, but this time, do full shading

- Advantage over deferred shading: less storage, less code modification, more general shading is possible, multisampling possible
- Disadvantage: needs to rasterize twice

So here, we actually do a second pass

78

    - Tile-based rendering



Tile-based rendering

- Problem: framebuffer is a lot of memory, especially with antialiasing
- Solution: render subsets of the screen at once
- For each tile of pixels
  - For each triangle
    - for each pixel

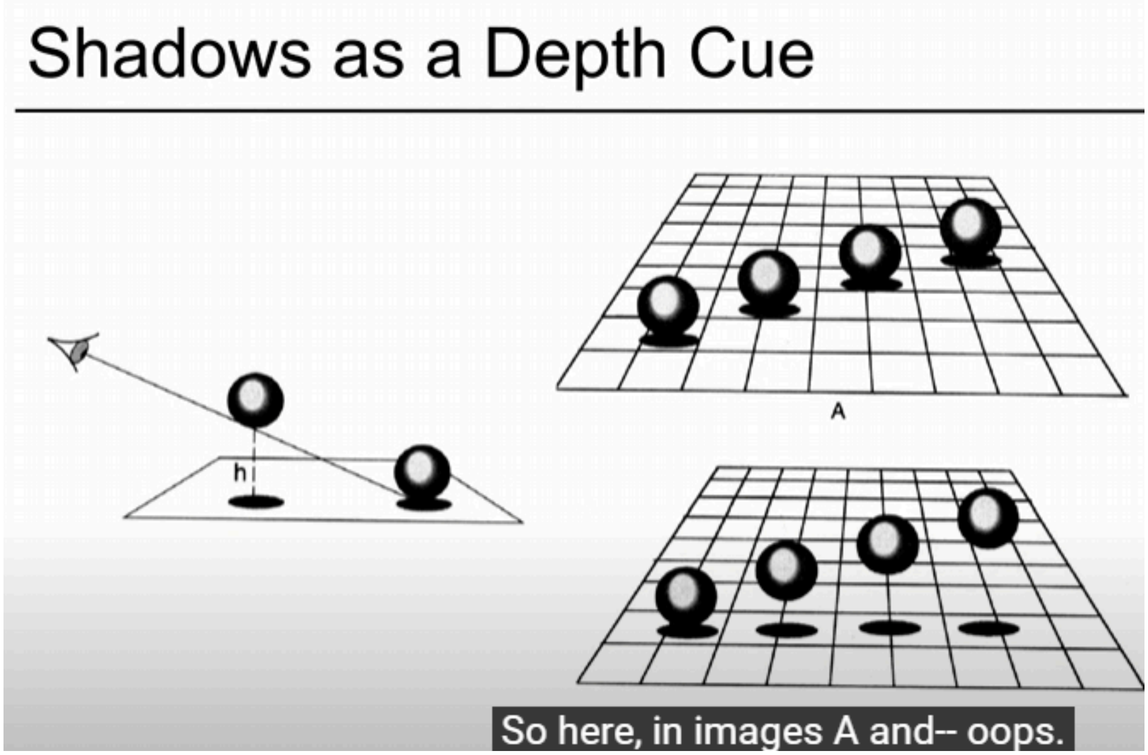
- Might need to handle a triangle in multiple tiles
  - redundant computation for projection and setup
- Used in mobile graphics cards

So one thing you could do is to render subsets of the screen

79

  - Shadows
  - Reflections
  - Global illumination
- **L19: Real-Time Shadows**

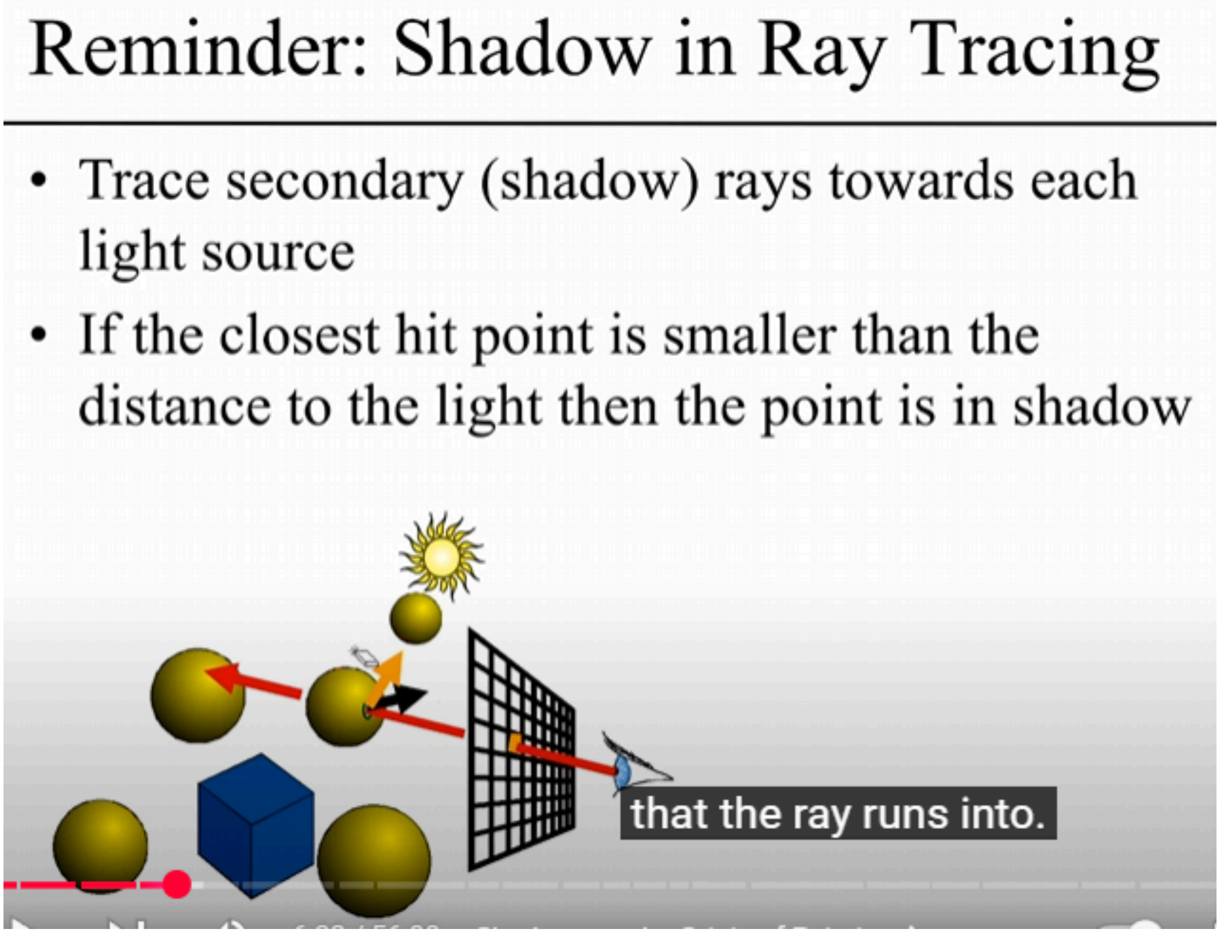
- Importance of Shadow
    - Depth cue

      Shadows as a Depth Cue

      So here, in images A and-- oops.
    - Scene Lighting
    - Realism
    - Contact Points

- Shadow in Ray Tracing

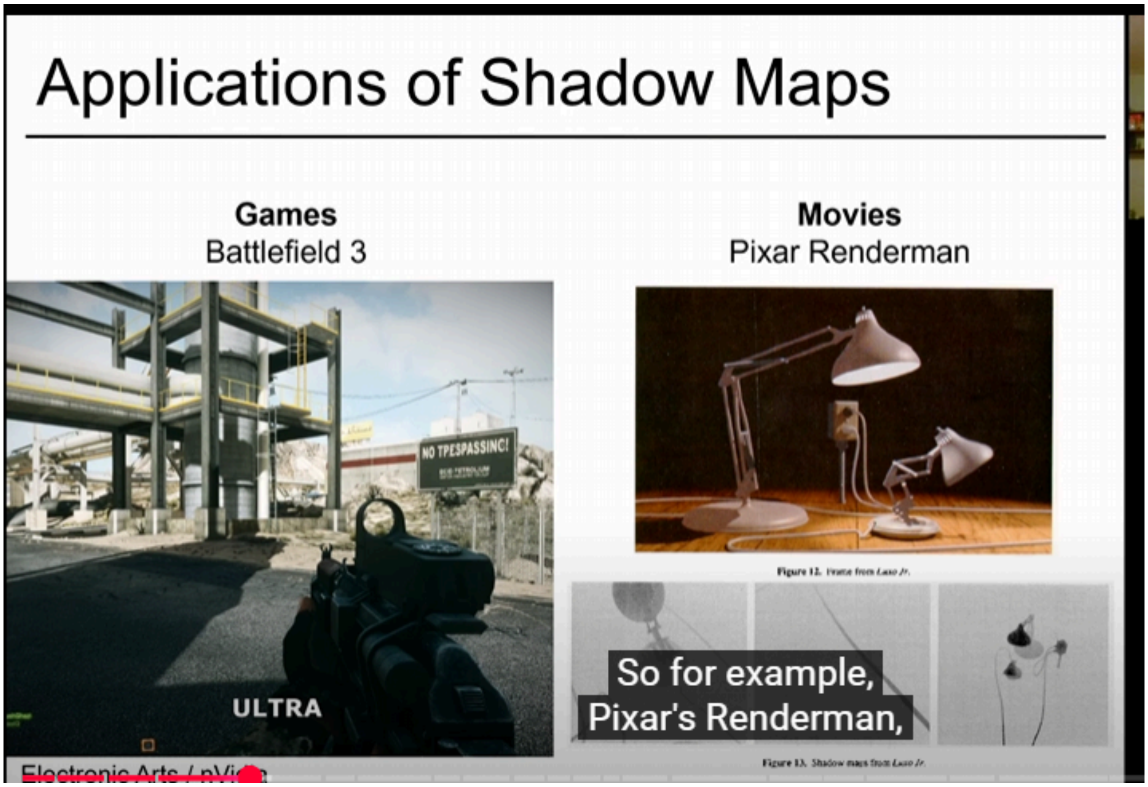


Reminder: Shadow in Ray Tracing

- Trace secondary (shadow) rays towards each light source
- If the closest hit point is smaller than the distance to the light then the point is in shadow

that the ray runs into.

- Shadow Maps
  - Example



Applications of Shadow Maps

Games
Battlefield 3

Movies
Pixar Renderman

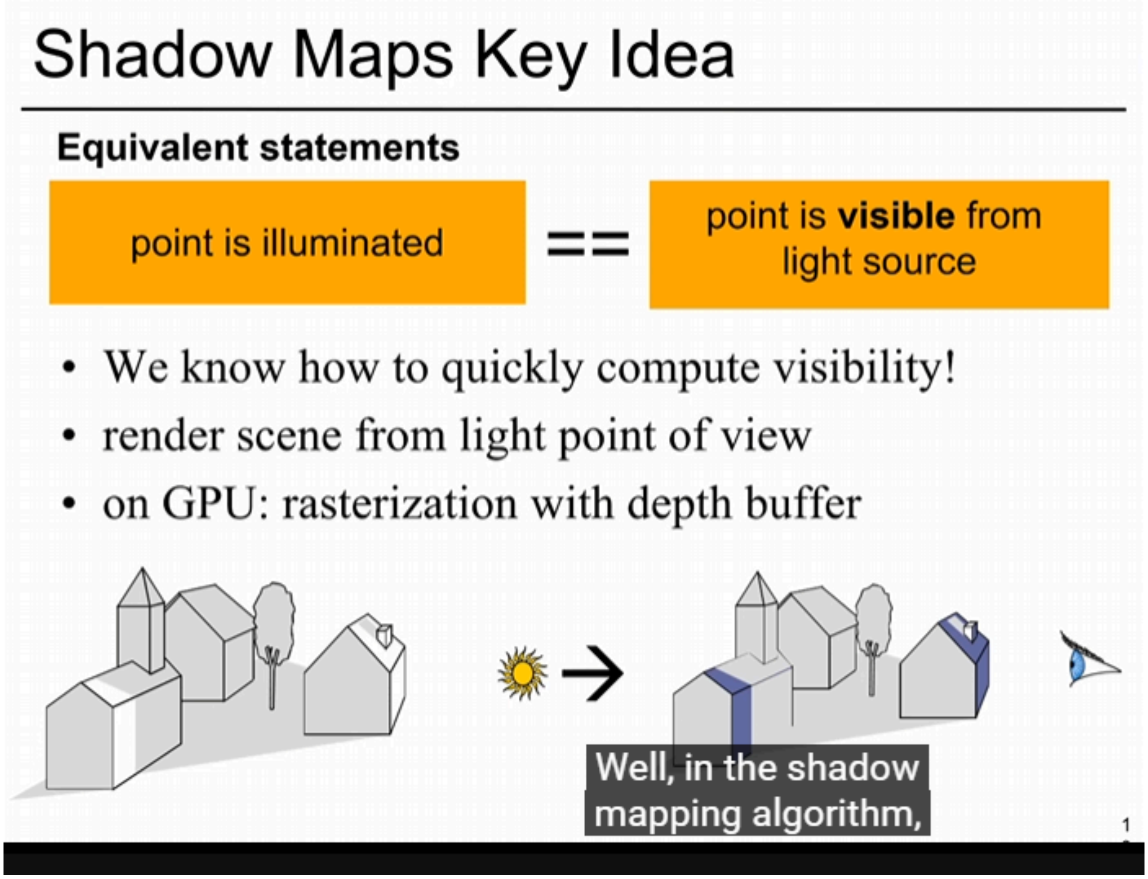
Figure 12. Frame from *Luxo Jr.*

Figure 13. Shadow maps from *Luxo Jr.*

So for example, Pixar's Renderman,

Electronic Arts / nVidia

- Key Idea



Shadow Maps Key Idea

Equivalent statements

point is illuminated == point is **visible** from light source

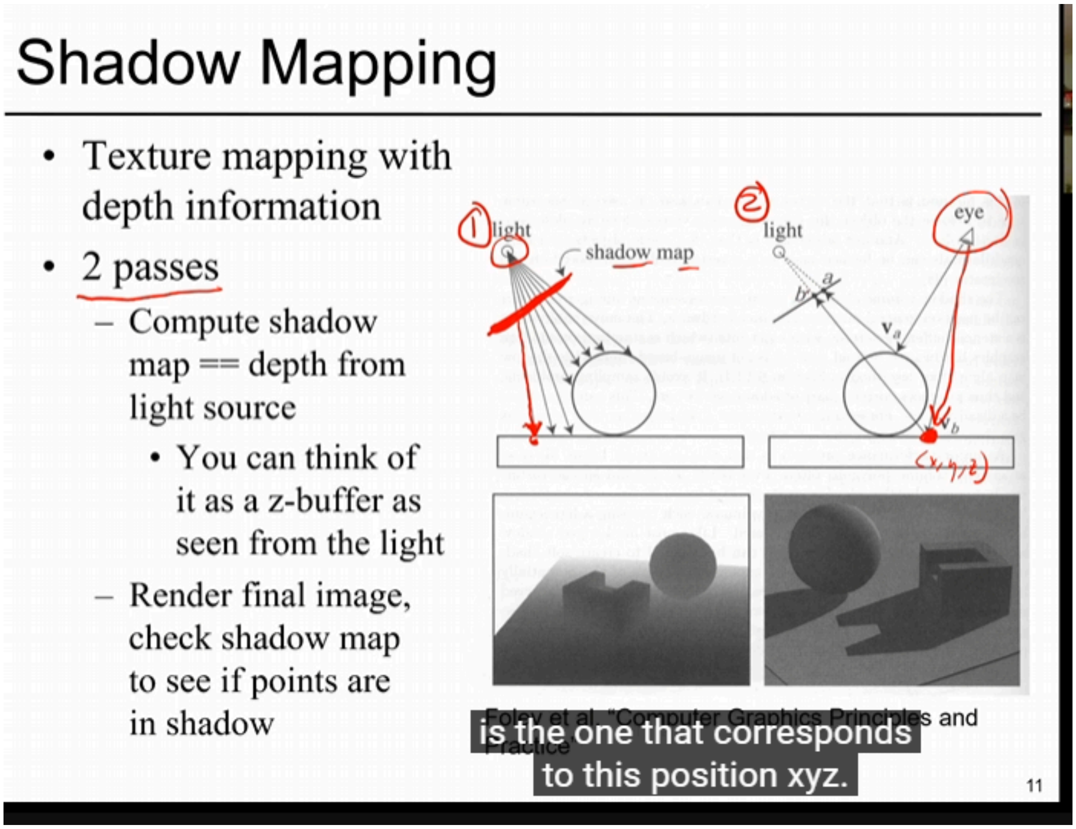
- We know how to quickly compute visibility!
- render scene from light point of view
- on GPU: rasterization with depth buffer

Well, in the shadow mapping algorithm,

1

  - Rasterize with the depth only to check if visible from the light source
    - By apply the camera position the the light source which can get z-buffer

- Compute the Shadow Map



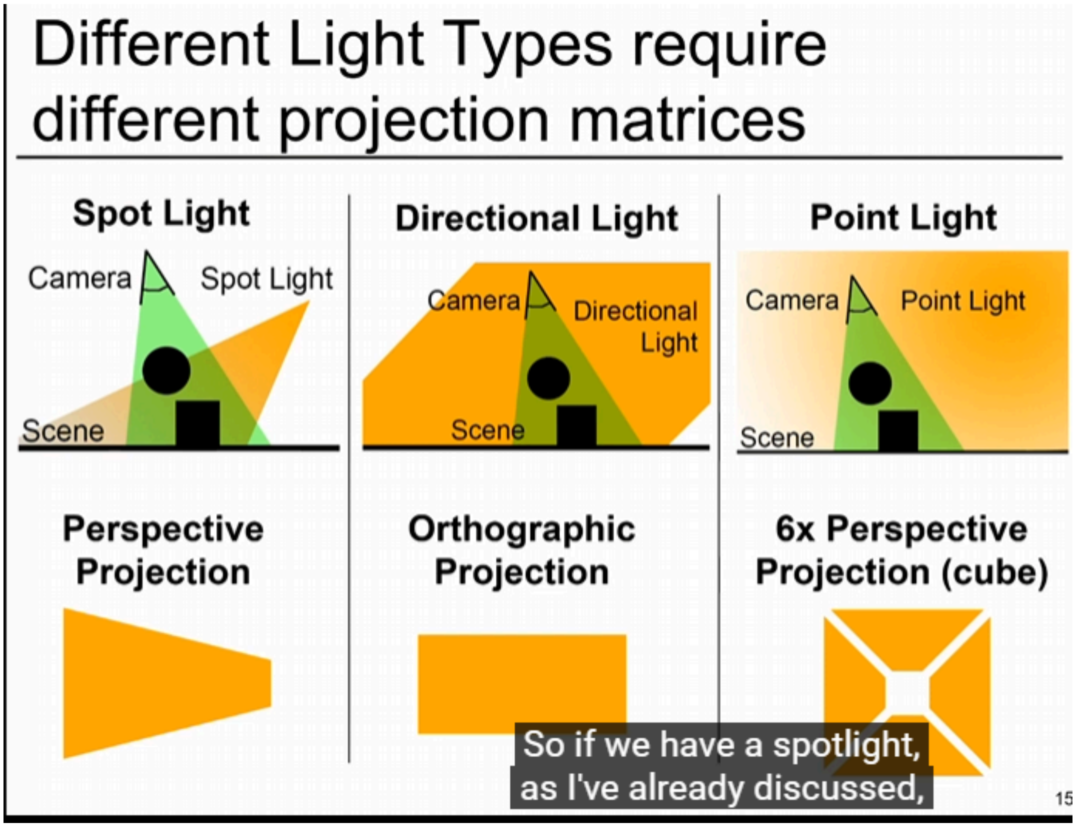
Shadow Mapping

- Texture mapping with depth information
- 2 passes
  - Compute shadow map == depth from light source
    - You can think of it as a z-buffer as seen from the light
  - Render final image, check shadow map to see if points are in shadow

Foley et al. "Computer Graphics Principles and Practice"

is the one that corresponds
to this position xyz.

- Different Light Types require different projection matrices

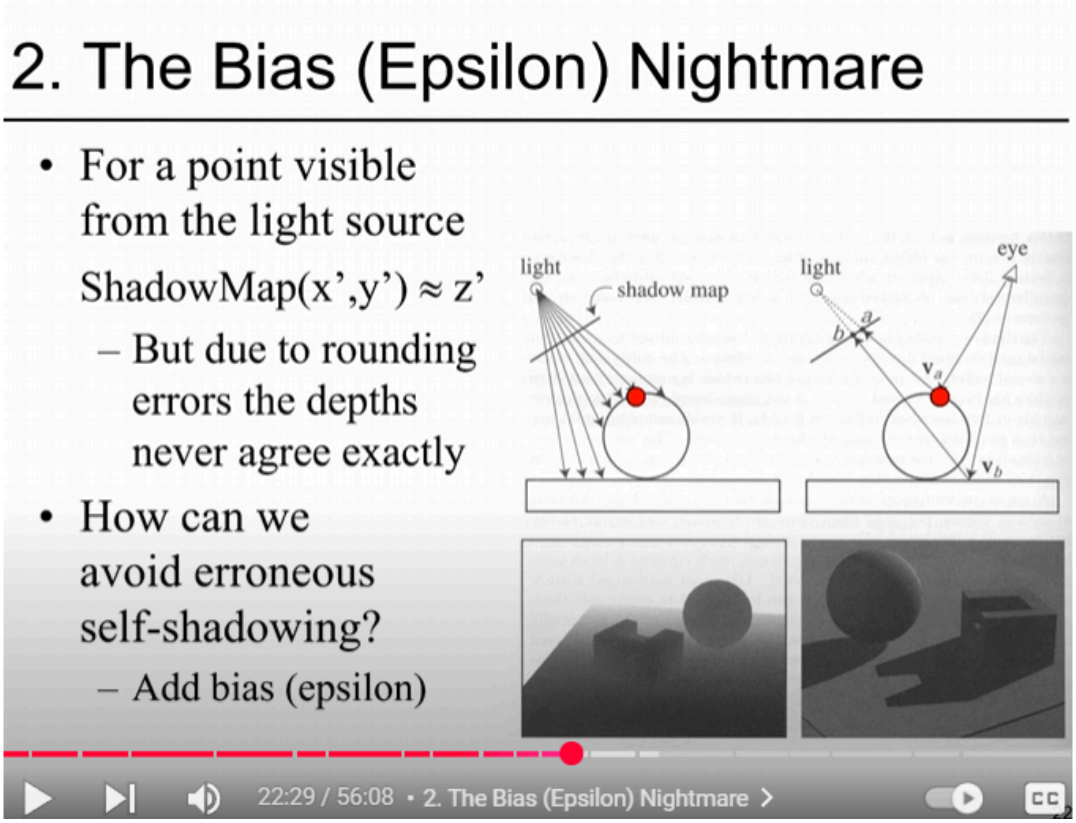


Different Light Types require different projection matrices

Spot Light	Directional Light	Point Light
Perspective Projection	Orthographic Projection	6x Perspective Projection (cube)

So if we have a spotlight,
as I've already discussed,

- The Bias (Epsilon) for Shadow Maps



2. The Bias (Epsilon) Nightmare

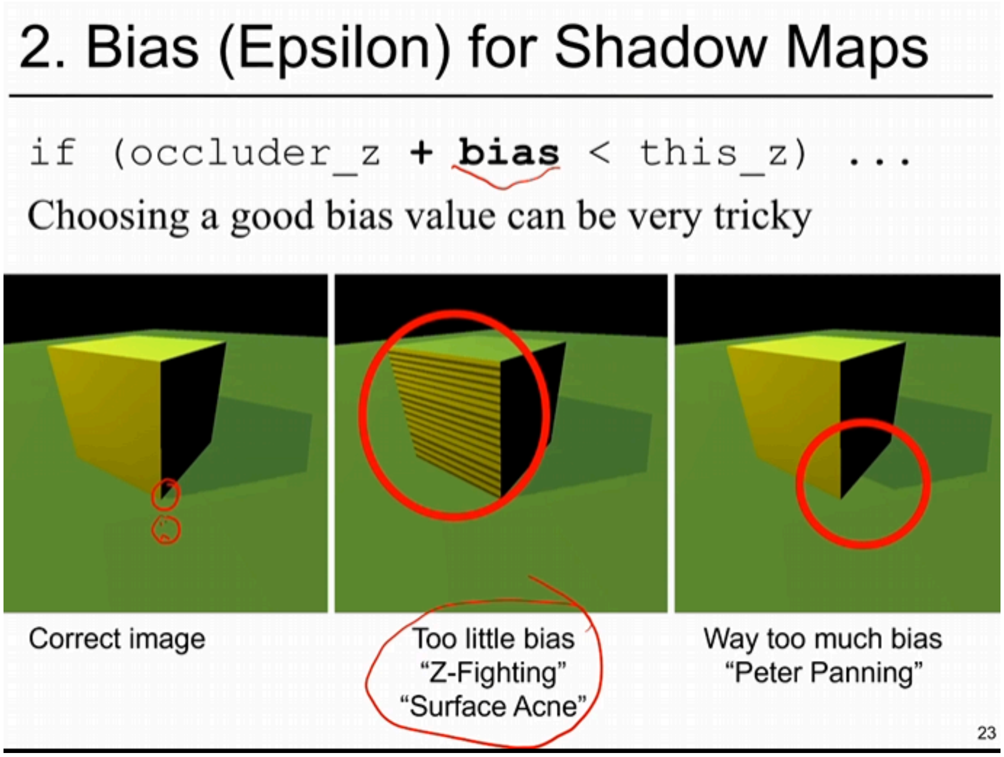
- For a point visible from the light source

  ShadowMap(x’,y’) ≈ z’

  - But due to rounding errors the depths never agree exactly
- How can we avoid erroneous self-shadowing?
  - Add bias (epsilon)

22:29 / 56:08 • 2. The Bias (Epsilon) Nightmare >

  - Example



2. Bias (Epsilon) for Shadow Maps

```
if (occluder_z + bias < this_z) ...
```

Choosing a good bias value can be very tricky

Correct image

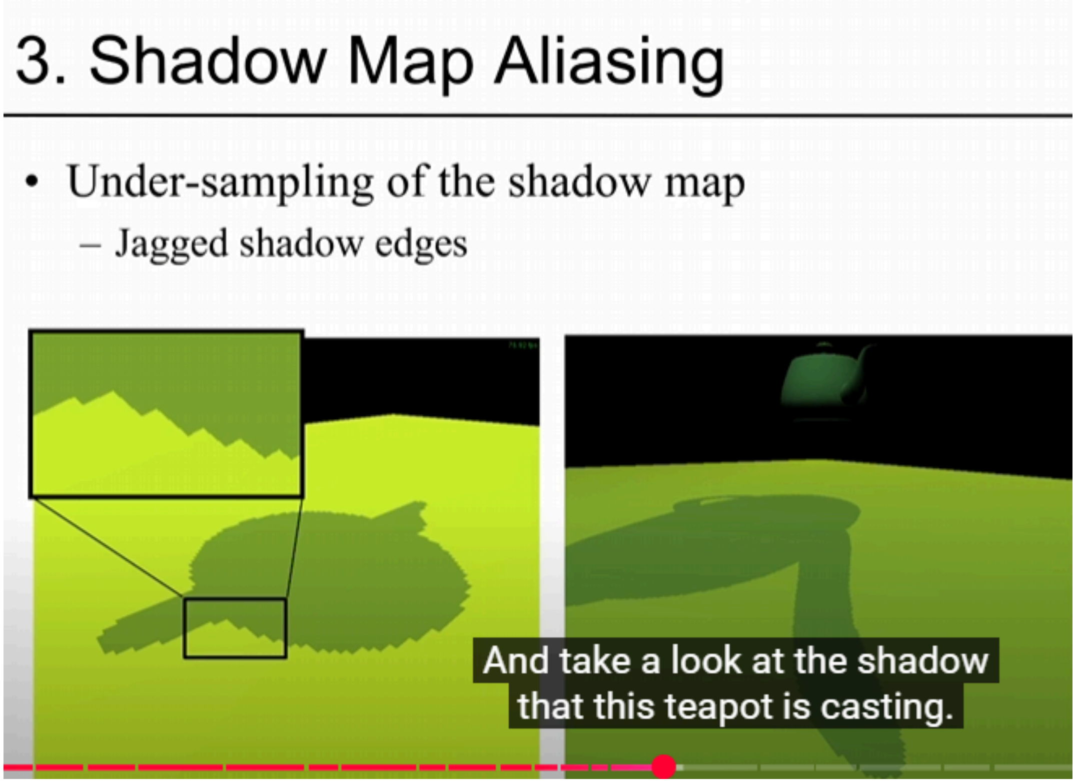
Too little bias
“Z-Fighting”
“Surface Acne”

Way too much bias
“Peter Panning”

23

  - for avoiding self shadow

- Shadow Map Aliasing

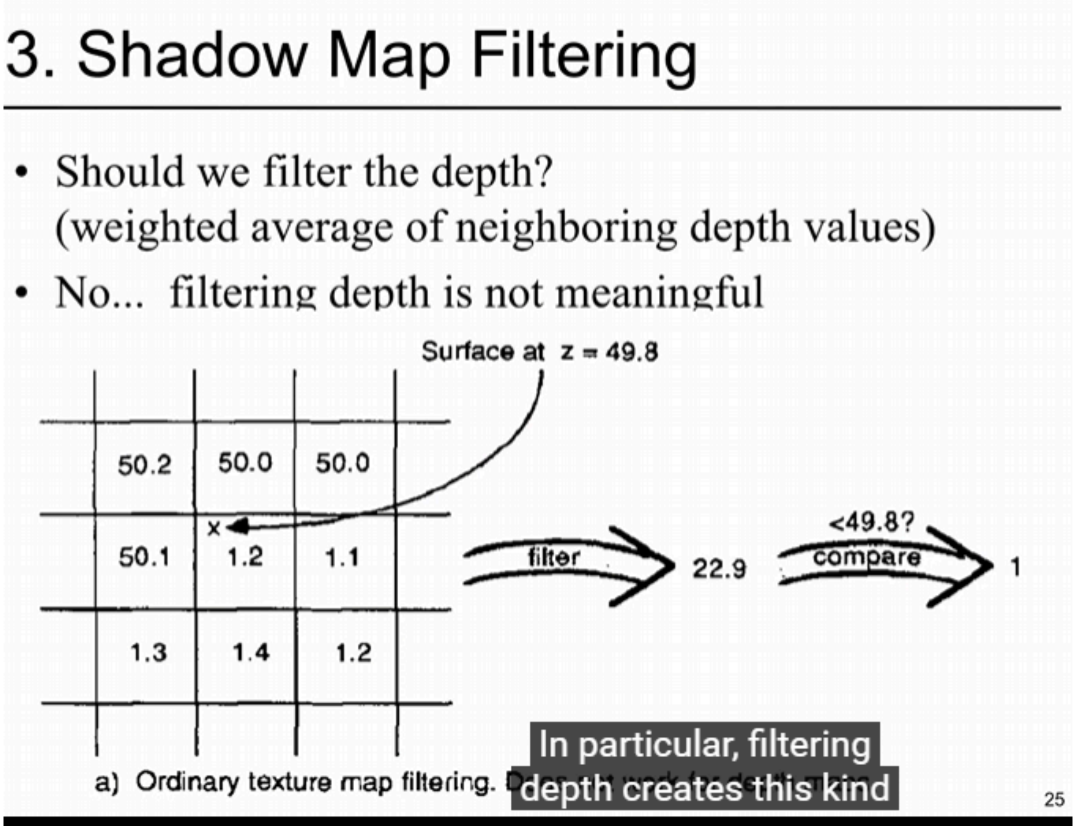


3. Shadow Map Aliasing

- Under-sampling of the shadow map
  - Jagged shadow edges

And take a look at the shadow that this teapot is casting.

- Shadow Map Filtering



3. Shadow Map Filtering

- Should we filter the depth? (weighted average of neighboring depth values)
- No... filtering depth is not meaningful

50.2	50.0	50.0
50.1	x 1.2	1.1
1.3	1.4	1.2

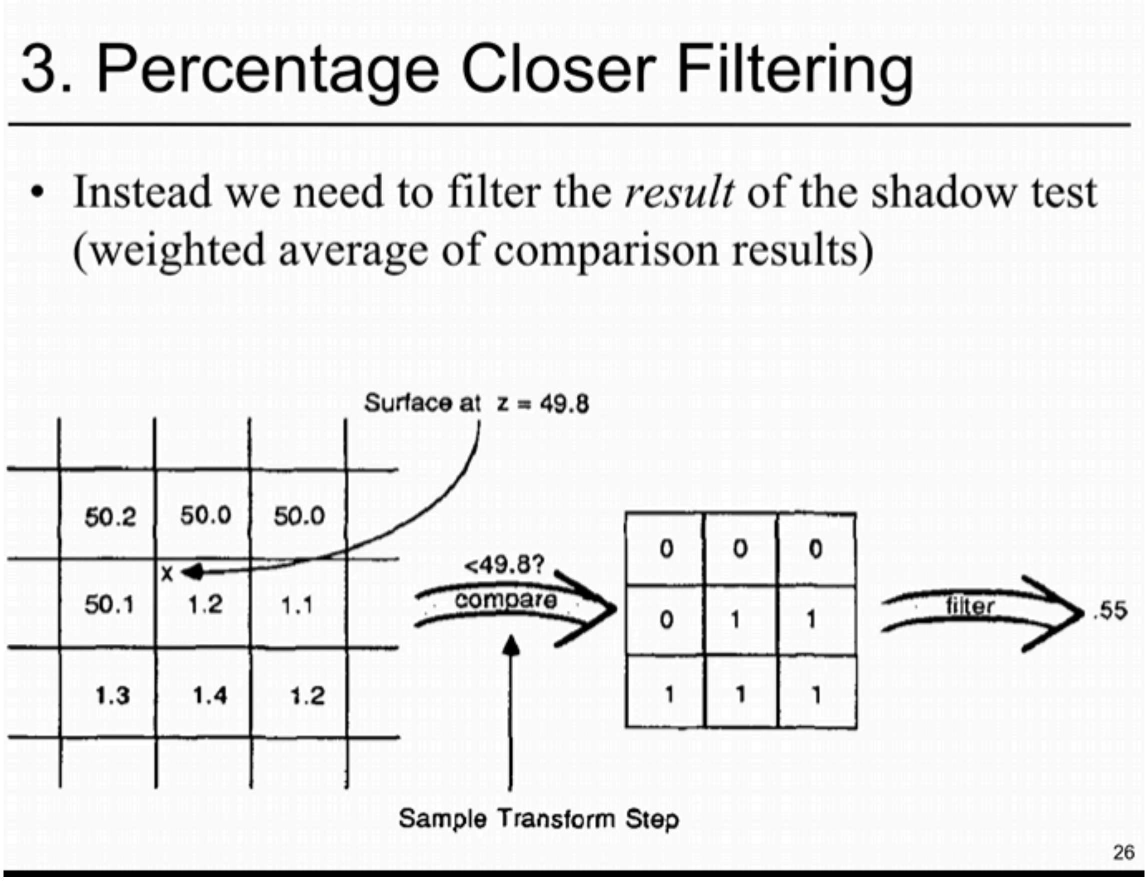
a) Ordinary texture map filtering. D[illegible]

In particular, filtering depth creates this kind

25

  - Does not make sense

- Percentage Closer Filtering



3. Percentage Closer Filtering

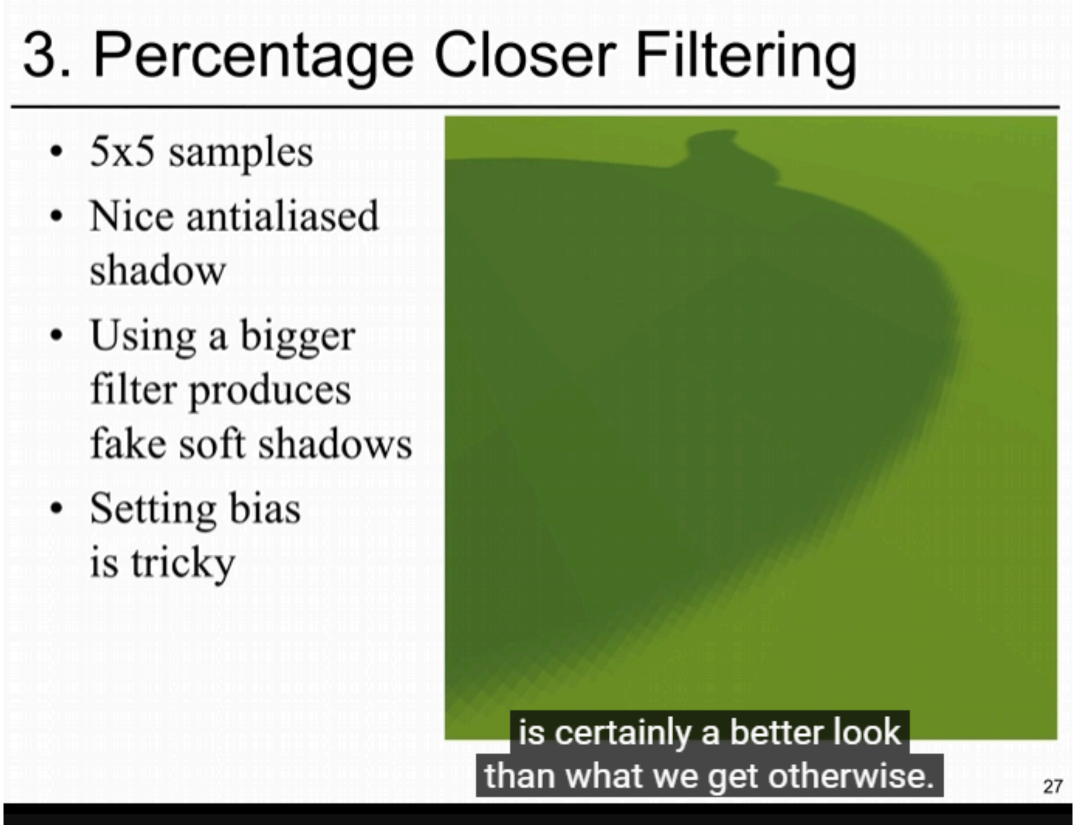
- Instead we need to filter the *result* of the shadow test (weighted average of comparison results)

50.2	50.0	50.0
50.1	1.2	1.1
1.3	1.4	1.2

0	0	0
0	1	1
1	1	1

26

  - Compute the pencentage of pixel which is occluded
  - Example



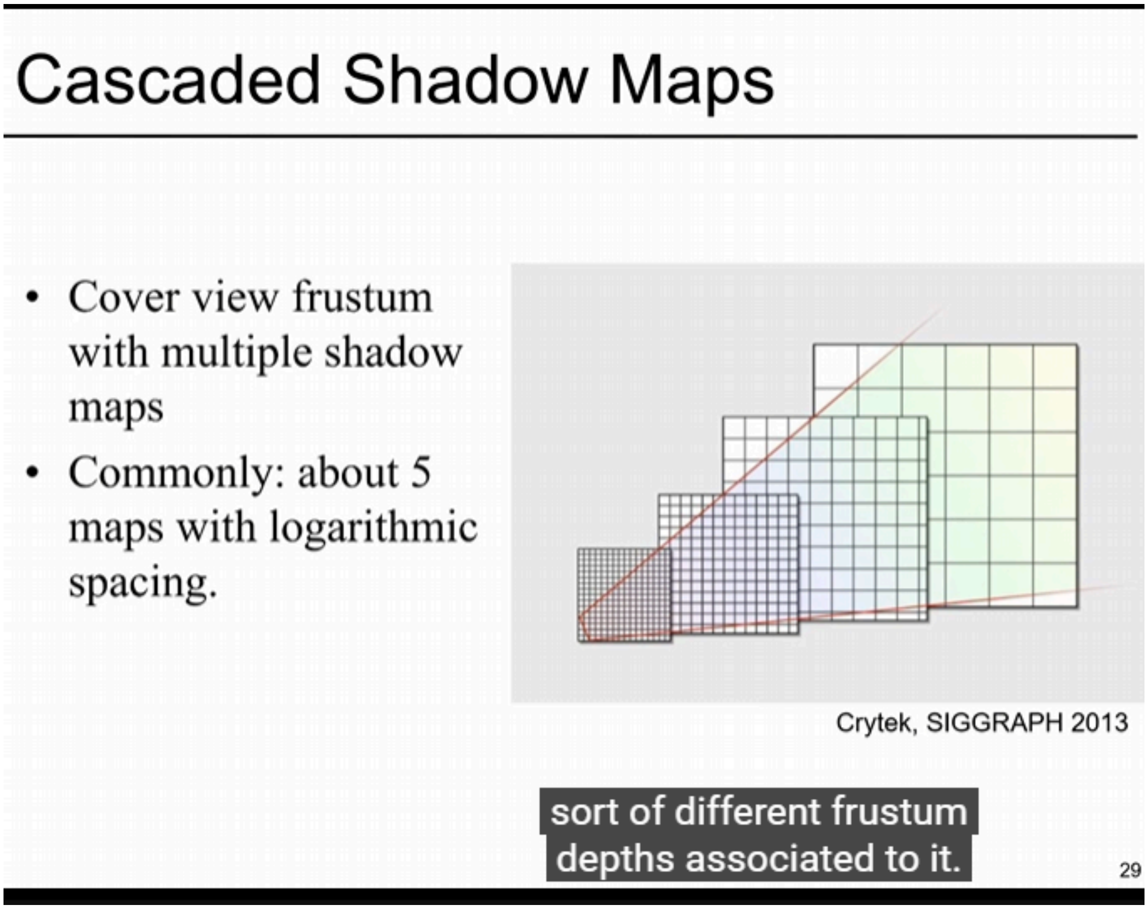
3. Percentage Closer Filtering

- 5x5 samples
- Nice antialiased shadow
- Using a bigger filter produces fake soft shadows
- Setting bias is tricky

is certainly a better look
than what we get otherwise.

27

- Cascaded Shadow Maps



Cascaded Shadow Maps

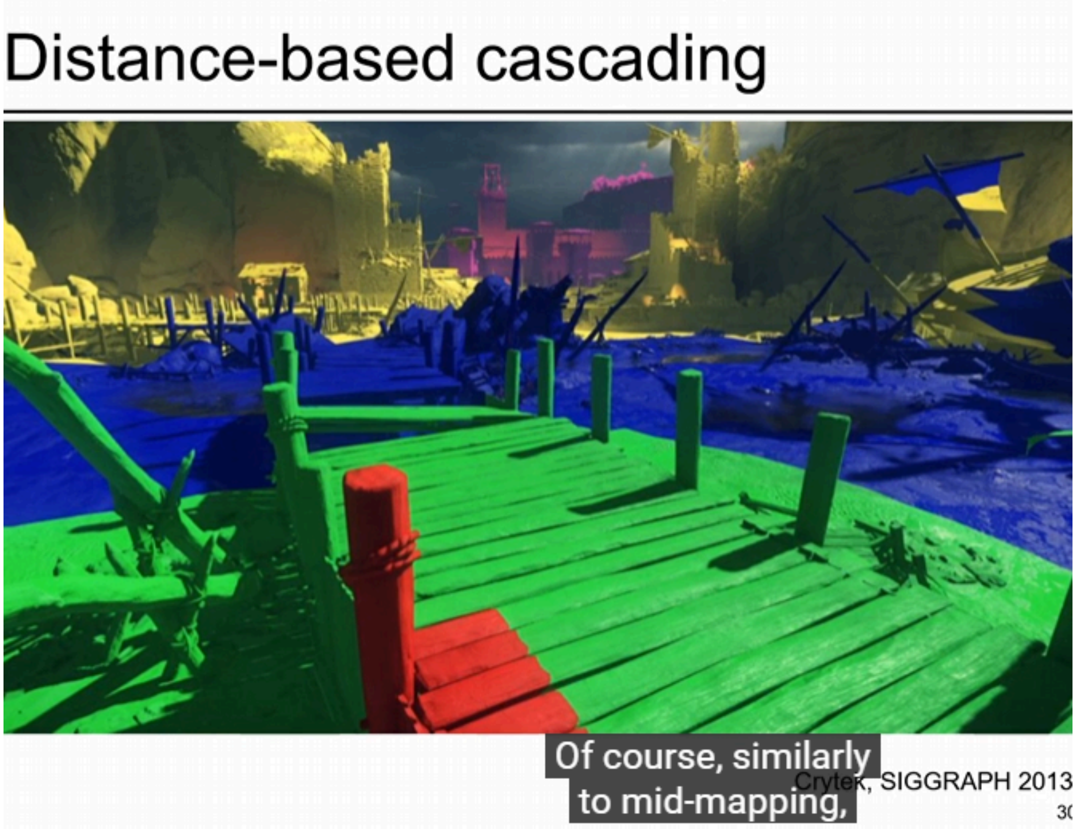
- Cover view frustum with multiple shadow maps
- Commonly: about 5 maps with logarithmic spacing.

Crytek, SIGGRAPH 2013

sort of different frustum depths associated to it.

29

  - Multiple depth shadow maps
  - Distance-base cascading



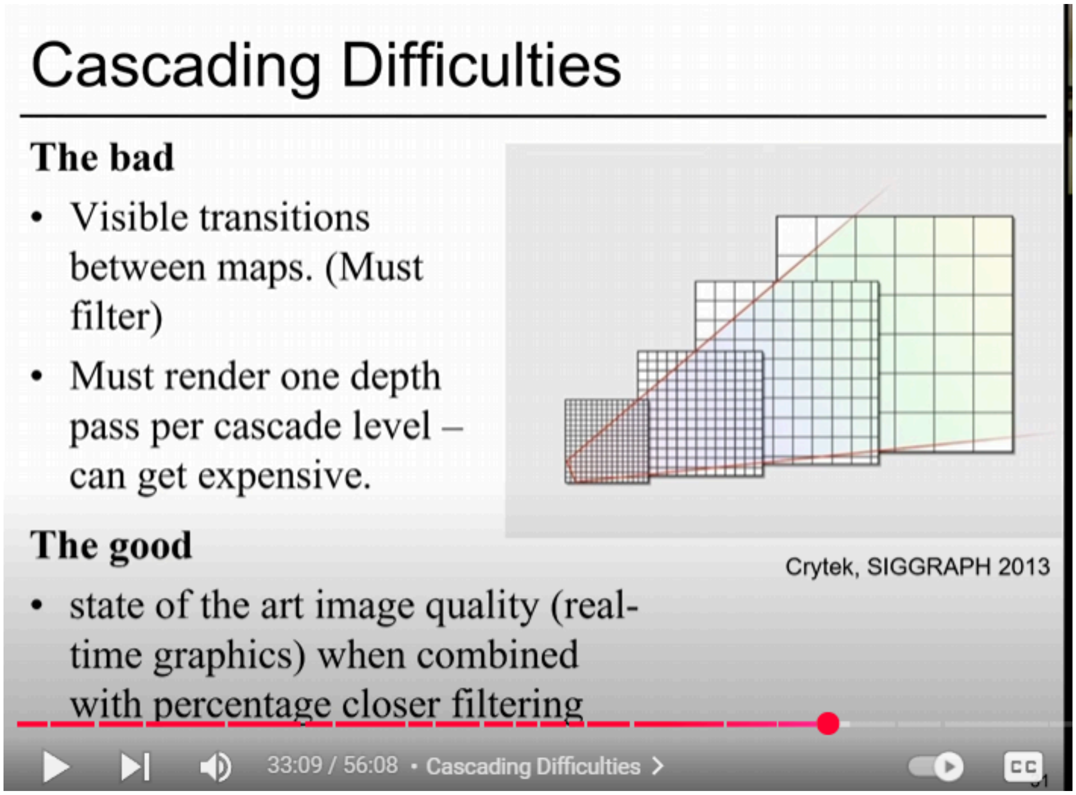
Distance-based cascading

Of course, similarly to mid-mapping,

Crytek, SIGGRAPH 2013

3(

- Pros and Cons



Cascading Difficulties

The bad

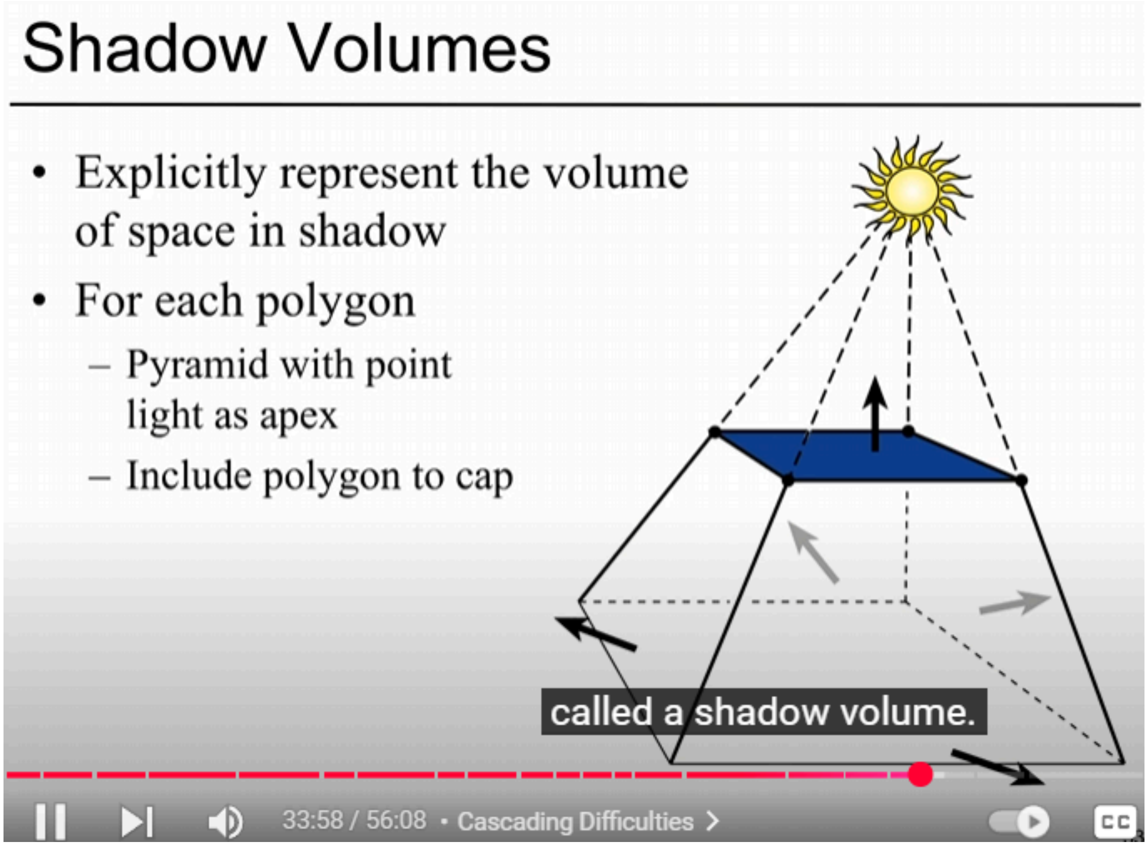
- Visible transitions between maps. (Must filter)
- Must render one depth pass per cascade level – can get expensive.

The good

- state of the art image quality (real-time graphics) when combined with percentage closer filtering

Crytek, SIGGRAPH 2013

- Shadow Volumes (Stencil Buffer)
  - Basic Idea

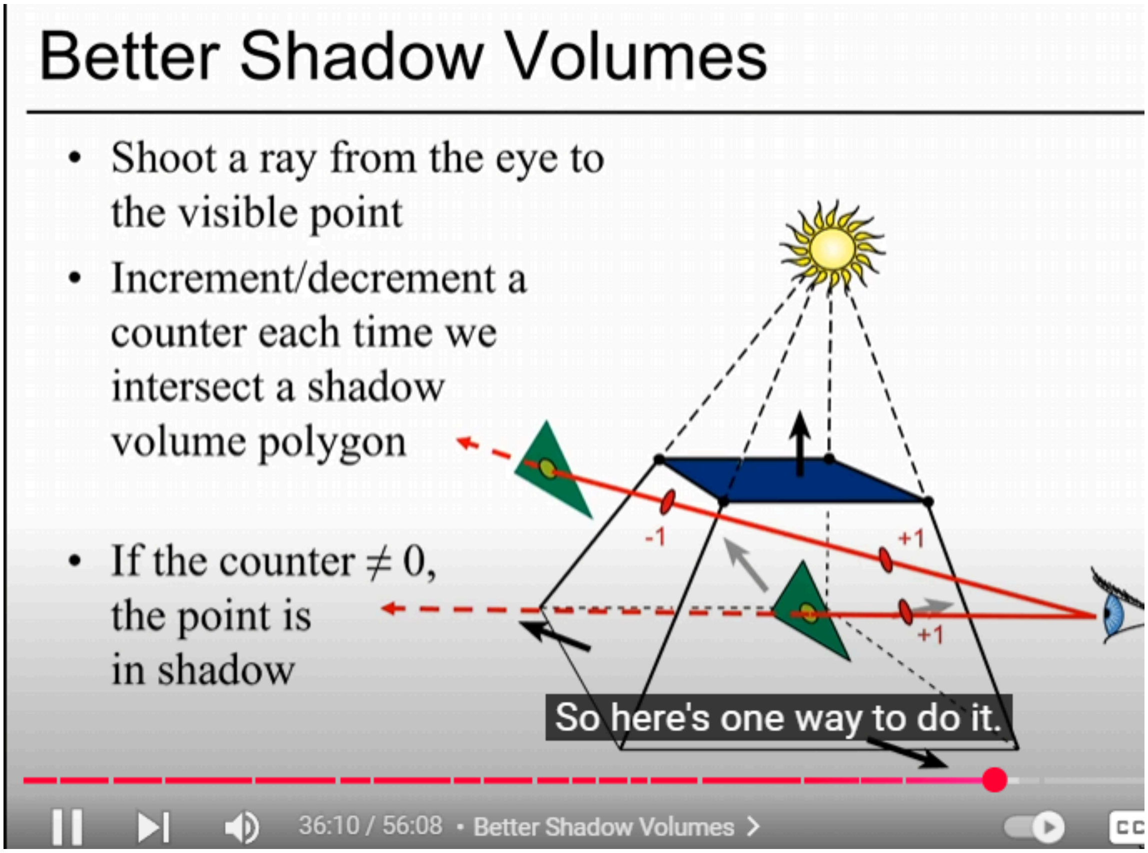


Shadow Volumes

- Explicitly represent the volume of space in shadow
- For each polygon
  - Pyramid with point light as apex
  - Include polygon to cap

- Create a shadow volume, check all object in the volume or not, if in, draw shadow, if not, lit it.

    - But very computational heavy
- Better Shadow Volumes

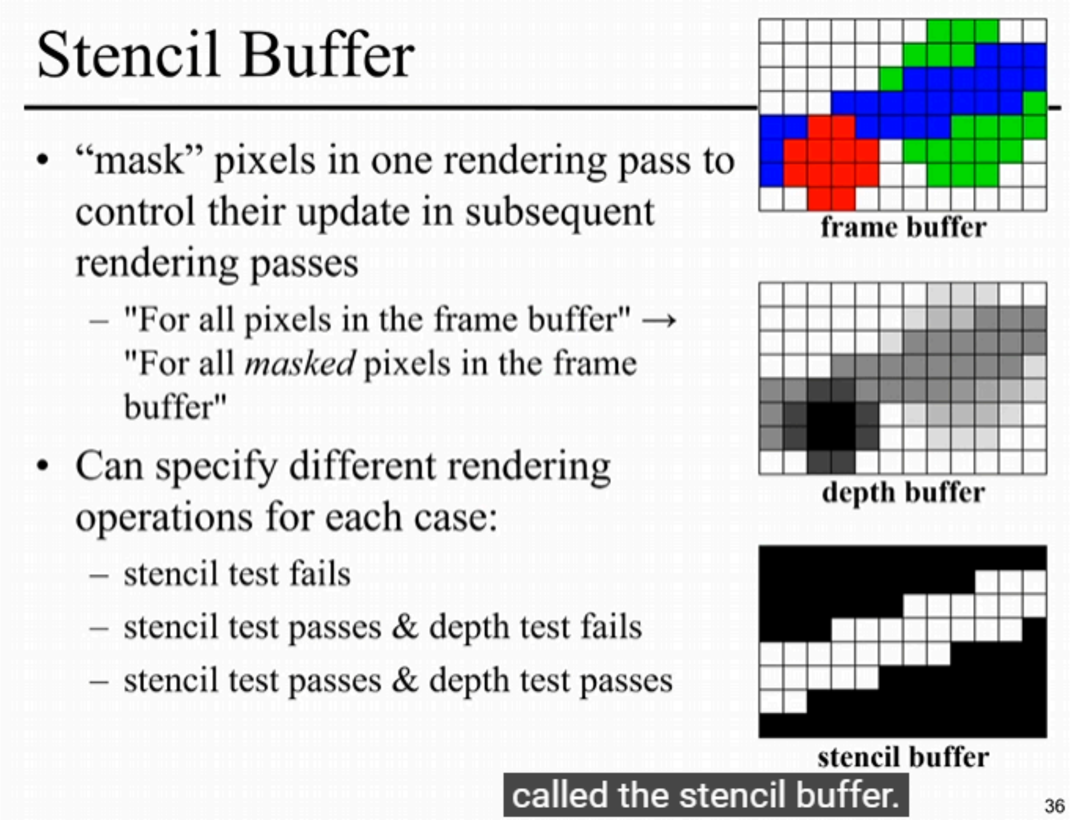


Better Shadow Volumes

- Shoot a ray from the eye to the visible point
- Increment/decrement a counter each time we intersect a shadow volume polygon
- If the counter ≠ 0, the point is in shadow

So here's one way to do it.

    - Stencil Buffer



Stencil Buffer

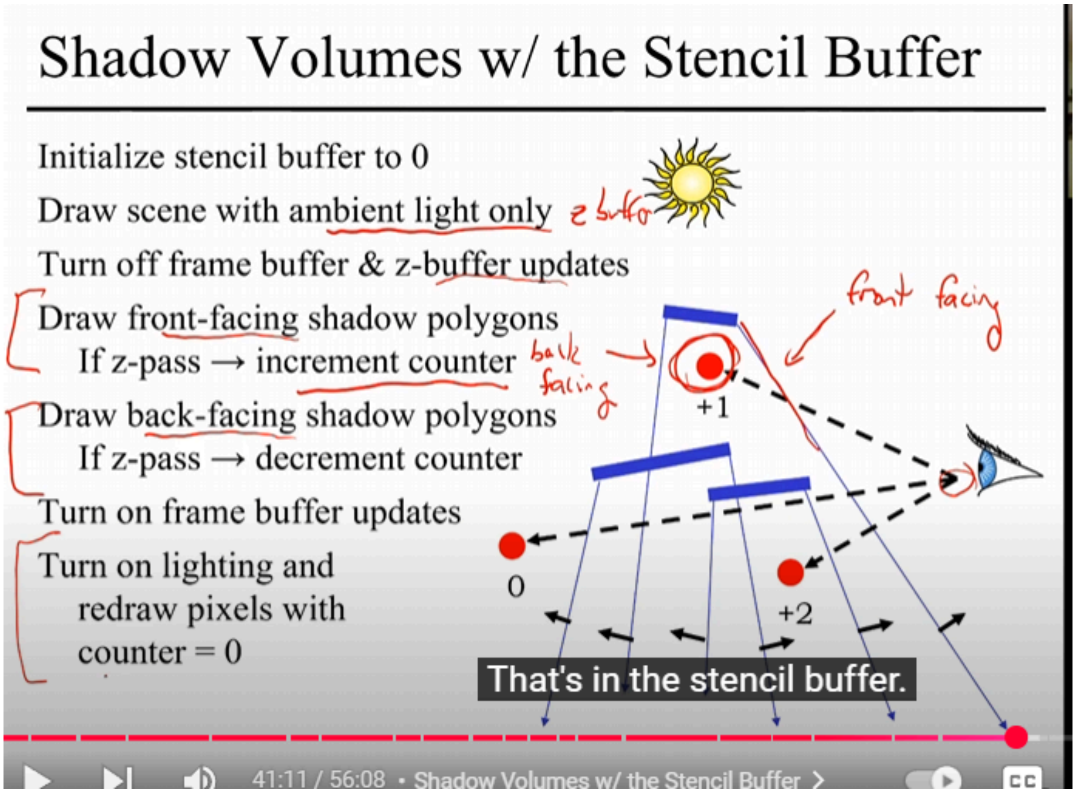
- "mask" pixels in one rendering pass to control their update in subsequent rendering passes
    - "For all pixels in the frame buffer" → "For all *masked* pixels in the frame buffer"
- Can specify different rendering operations for each case:
    - stencil test fails
    - stencil test passes & depth test fails
    - stencil test passes & depth test passes

called the stencil buffer.

36

        - unprecise z-buffer

- Shadow Volumes with the Stencil Buffer



Shadow Volumes w/ the Stencil Buffer

Initialize stencil buffer to 0

Draw scene with ambient light only z buffer

Turn off frame buffer & z-buffer updates

Draw front-facing shadow polygons
    If z-pass → increment counter

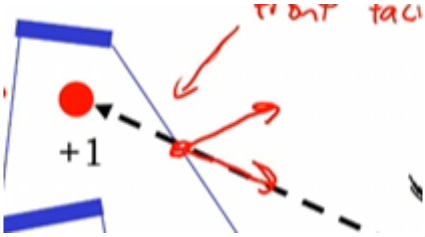
Draw back-facing shadow polygons
    If z-pass → decrement counter

Turn on frame buffer updates

Turn on lighting and
    redraw pixels with
    counter = 0

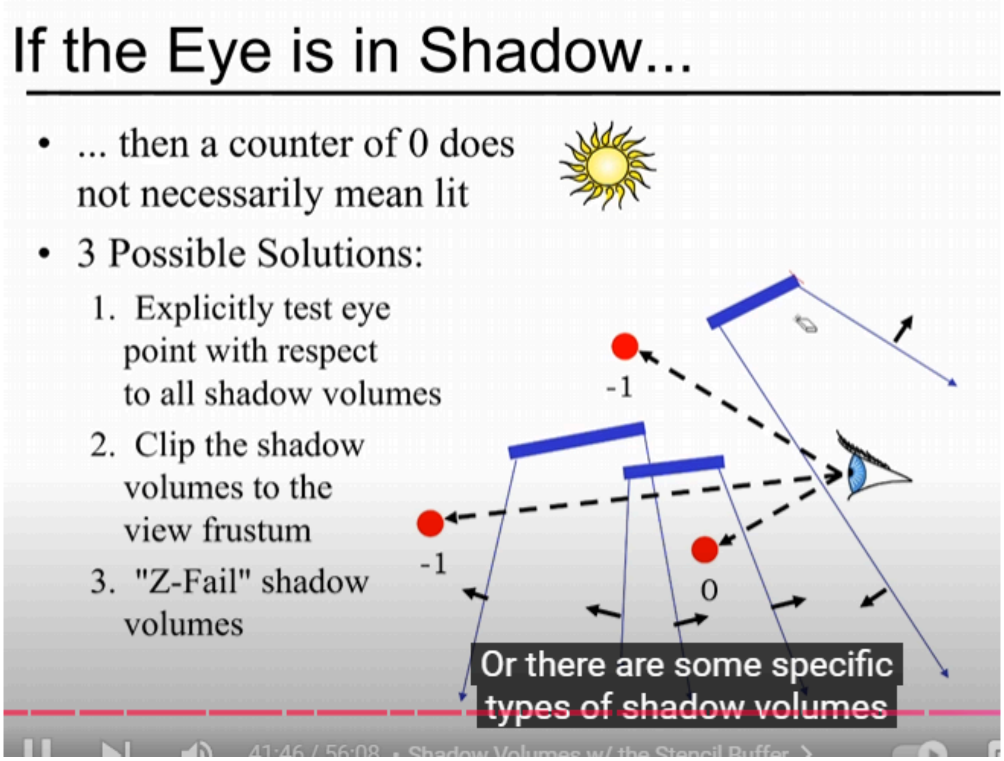
That's in the stencil buffer.

  - Calculate the dot product with normal and the direction to the eye

    , if positive, then it is front facing, if negative, then it is back facing. apply the increment/decrement counter again. draw the lighting with counter = 0

- Solutions if eye in the shadow

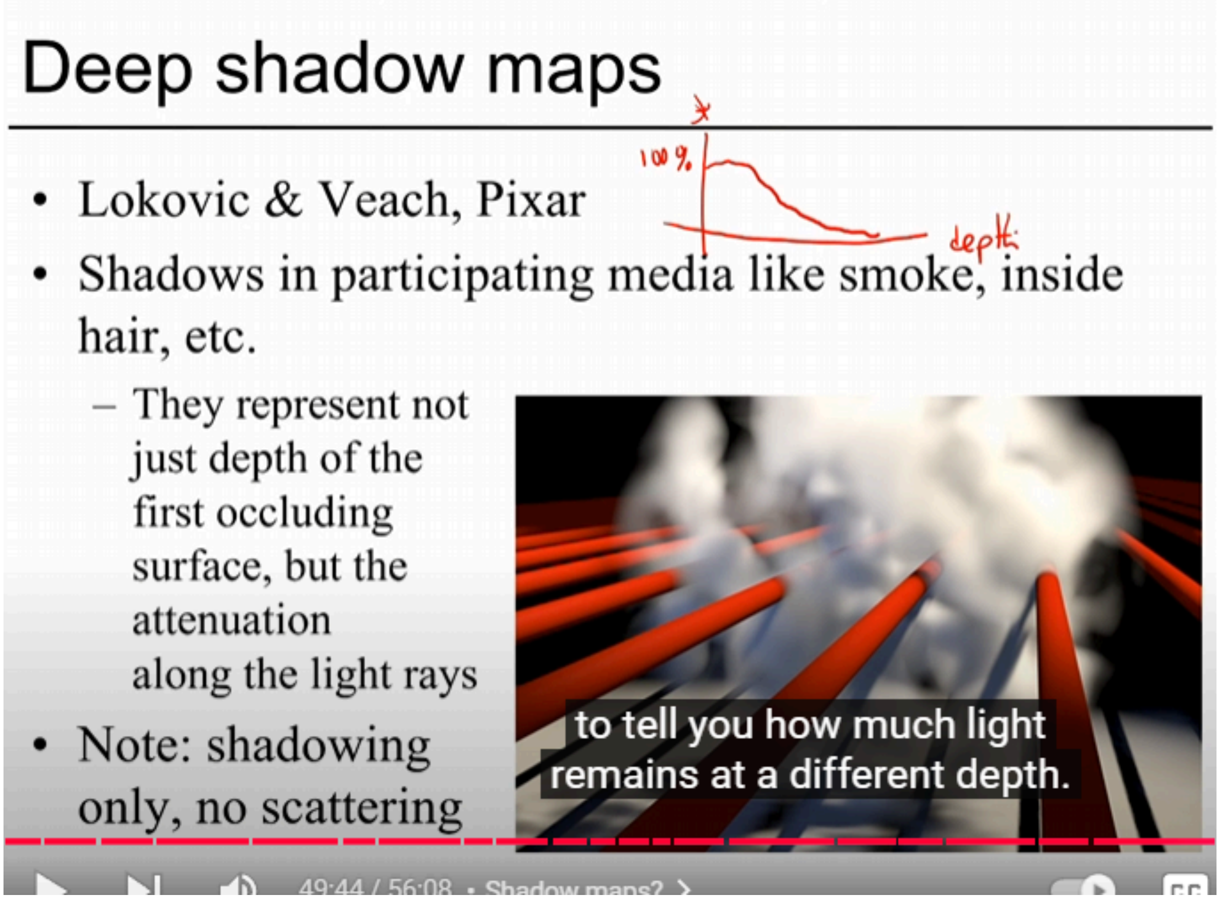


If the Eye is in Shadow...

- ... then a counter of 0 does not necessarily mean lit
- 3 Possible Solutions:
  1. Explicitly test eye point with respect to all shadow volumes
  2. Clip the shadow volumes to the view frustum
  3. "Z-Fail" shadow volumes

Or there are some specific types of shadow volumes

- Deep Shadow Maps



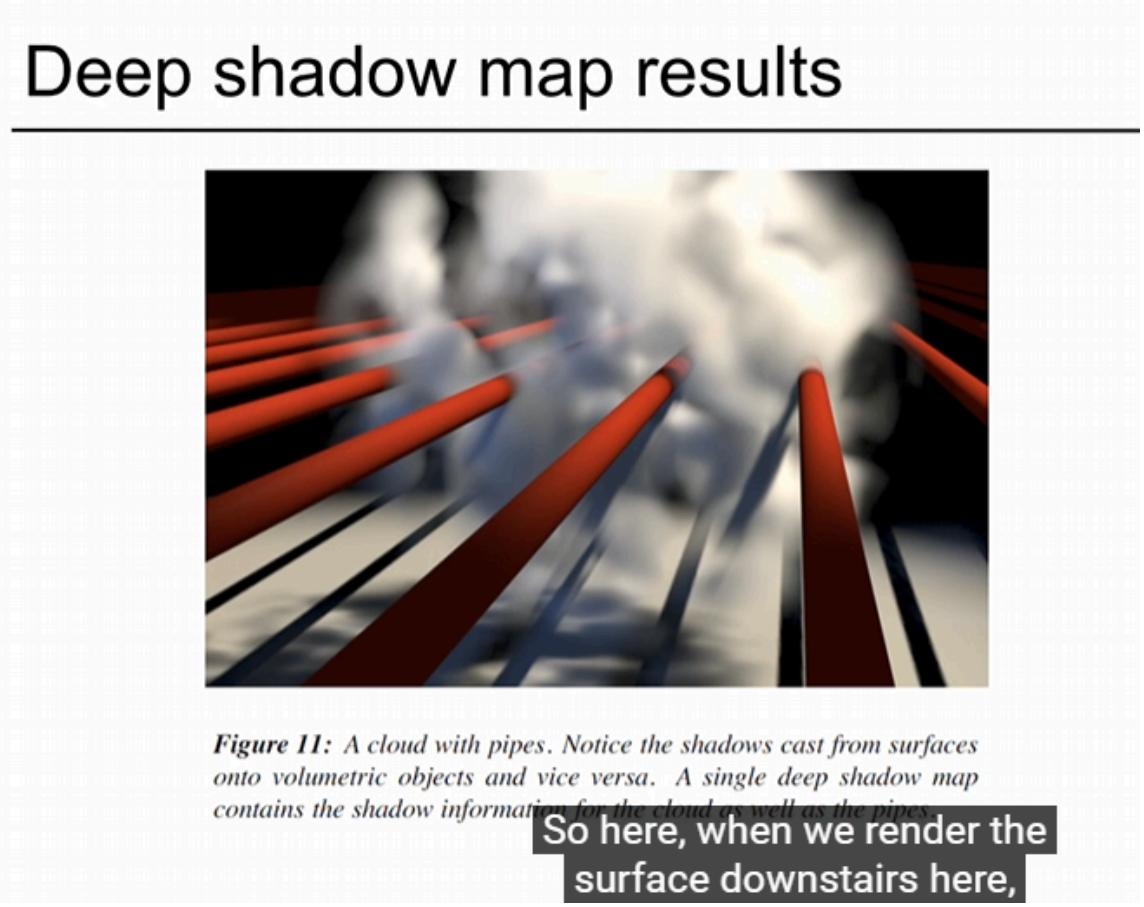
Deep shadow maps

- Lokovic & Veach, Pixar
- Shadows in participating media like smoke, inside hair, etc.
  - They represent not just depth of the first occluding surface, but the attenuation along the light rays
- Note: shadowing only, no scattering

to tell you how much light remains at a different depth.

- for volumetric effect, semi-transparenet object, small occluders

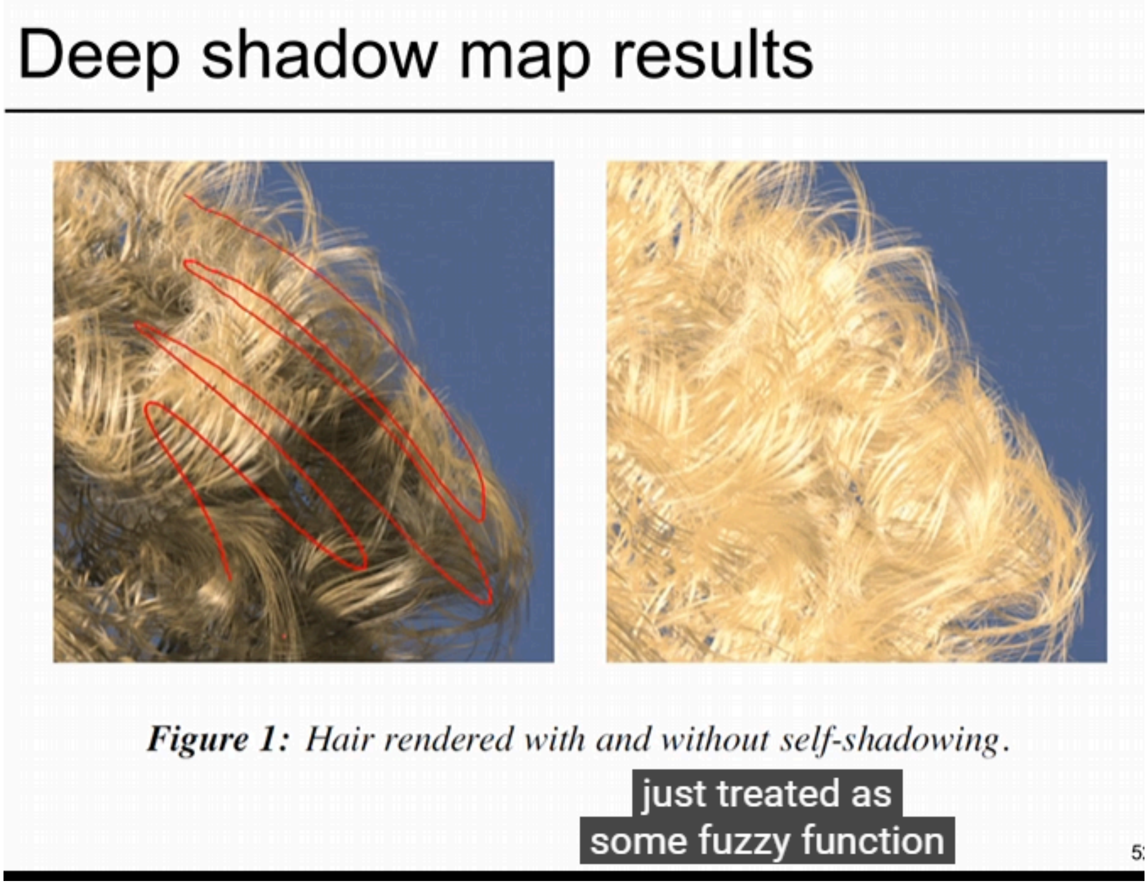
- Results



Deep shadow map results

Figure 11: *A cloud with pipes. Notice the shadows cast from surfaces onto volumetric objects and vice versa. A single deep shadow map contains the shadow information for the cloud as well as the pipes.*

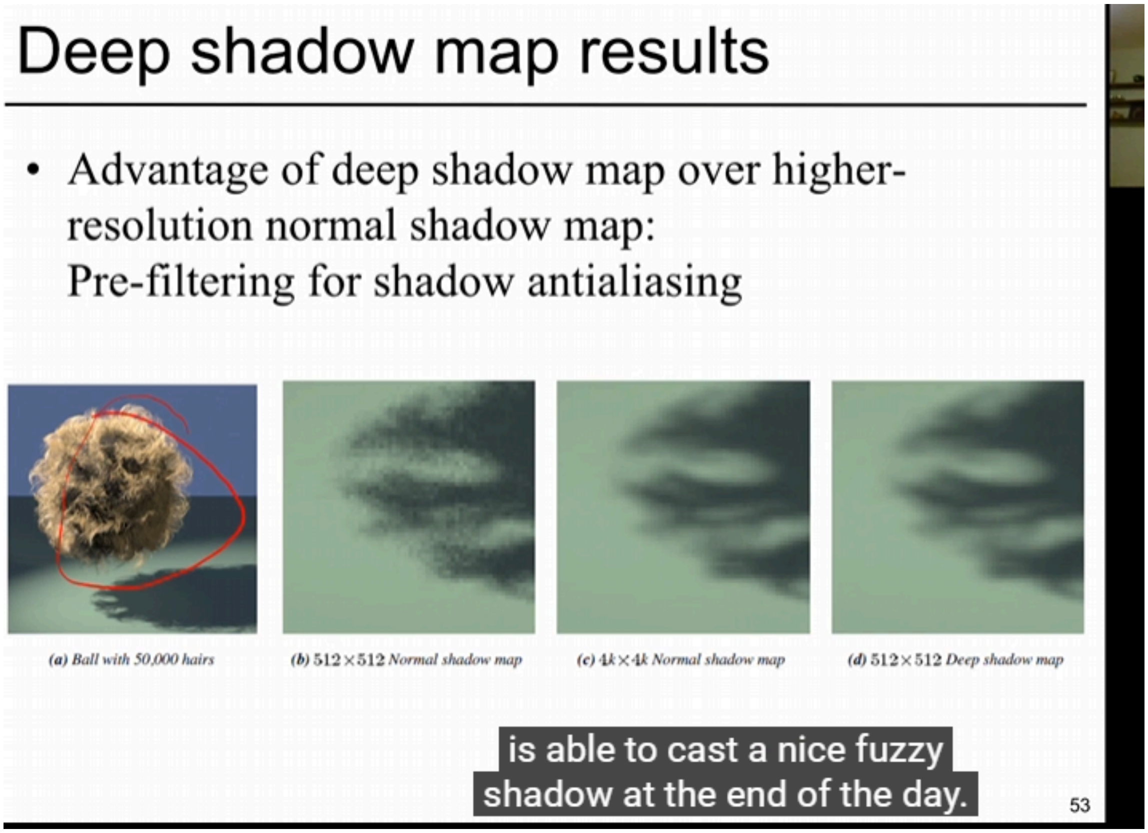
So here, when we render the surface downstairs here,



Deep shadow map results

Figure 1: *Hair rendered with and without self-shadowing.*

just treated as some fuzzy function



Deep shadow map results

- Advantage of deep shadow map over higher-resolution normal shadow map: Pre-filtering for shadow antialiasing

(a) Ball with 50,000 hairs

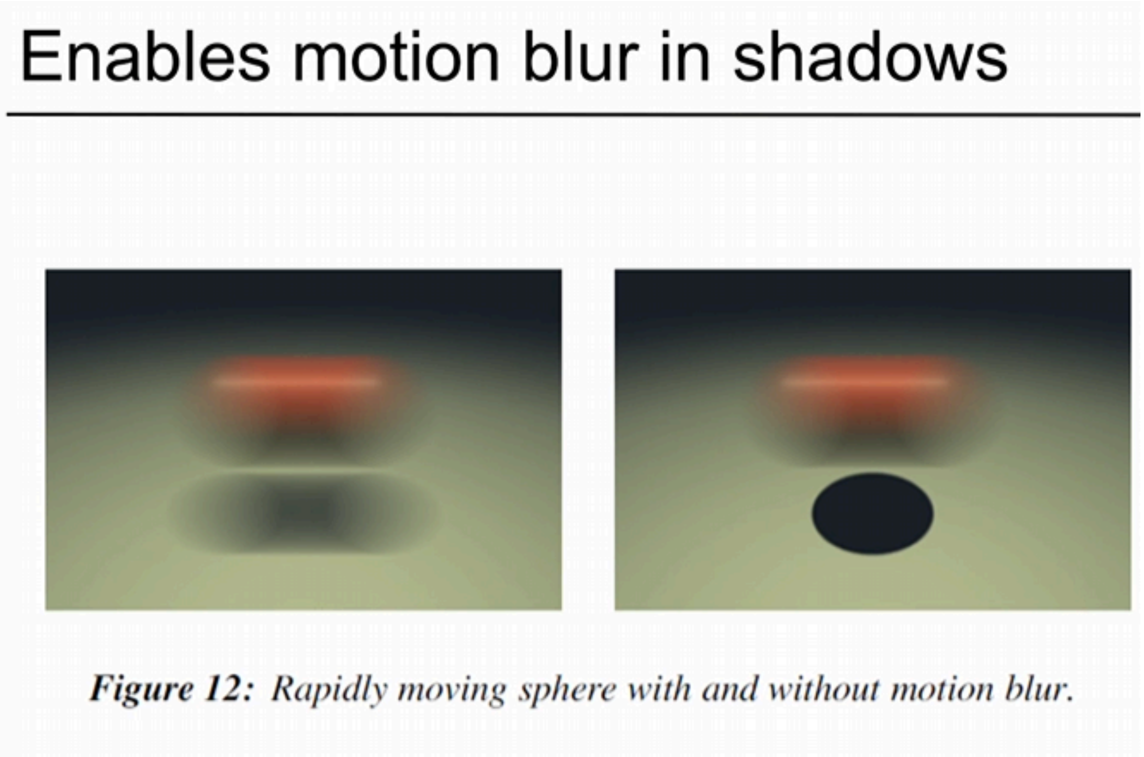
(b) 512×512 Normal shadow map

(c) 4k×4k Normal shadow map

(d) 512×512 Deep shadow map

is able to cast a nice fuzzy shadow at the end of the day.

53

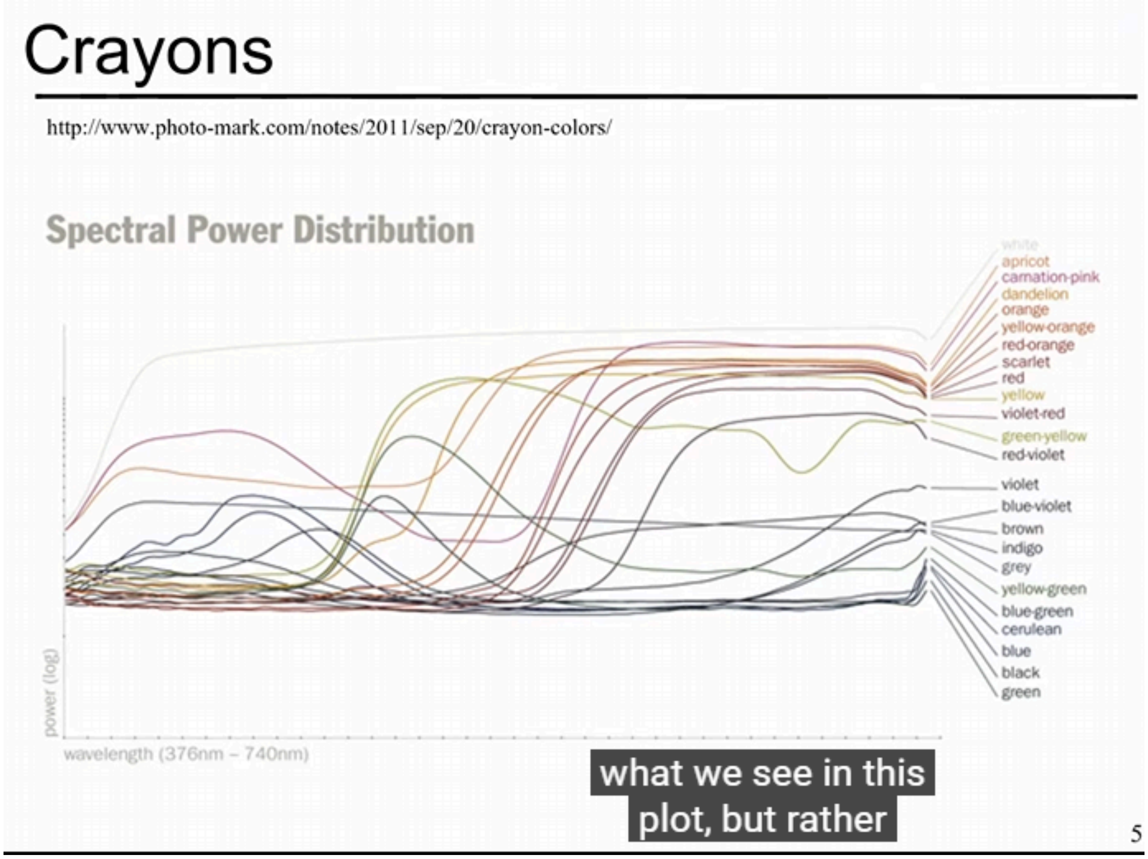


Enables motion blur in shadows

Figure 12: *Rapidly moving sphere with and without motion blur.*

- **L20: Color**
    - Spectra

  - Crayons



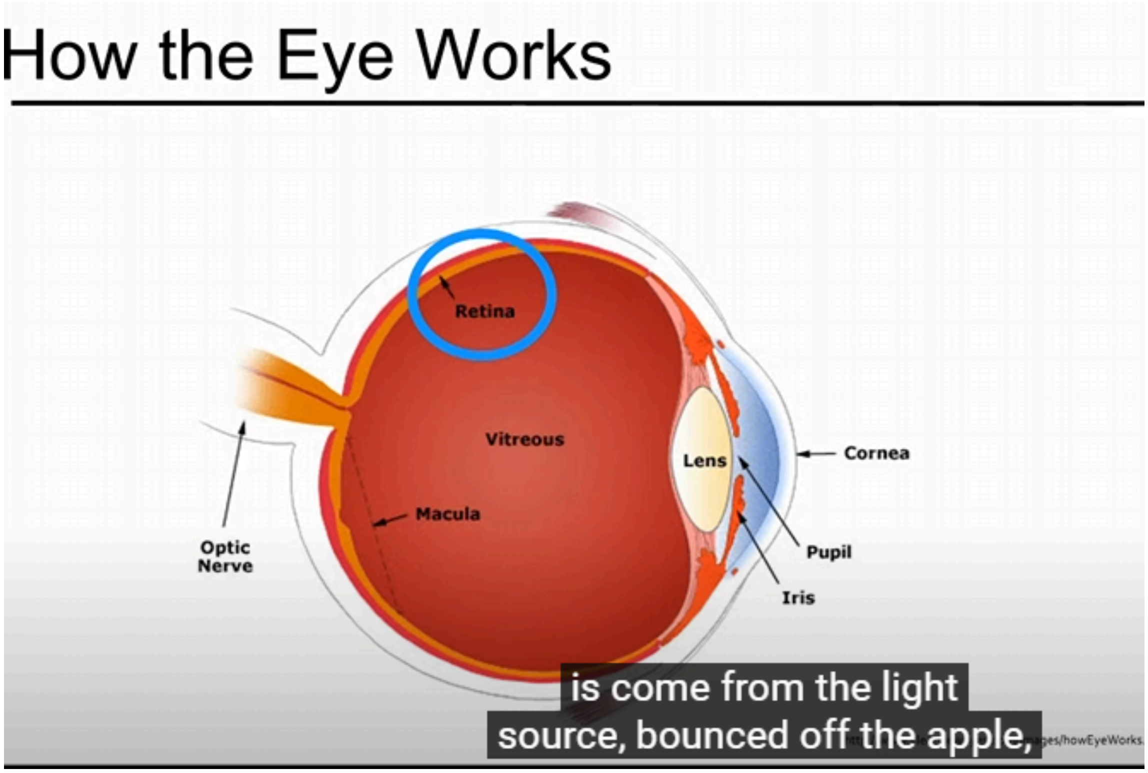
Crayons

http://www.photo-mark.com/notes/2011/sep/20/crayon-colors/

what we see in this plot, but rather

5

- Cones and spectral response
  - How the Eye Works

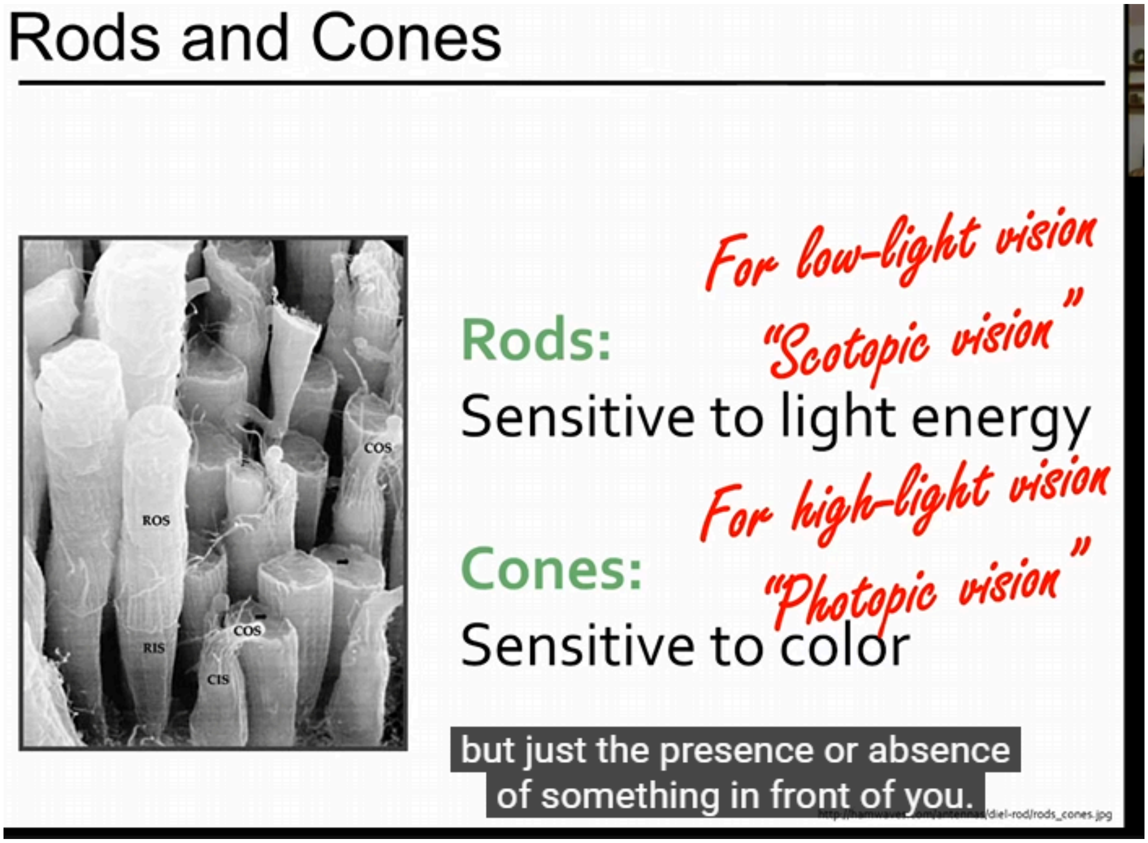


How the Eye Works

is come from the light source, bounced off the apple,

    - Photon go through Cornea, Lens, Virtreous, finally to Retina, Retina perceive light signal and convert to biological signal.

- Retina Element



Rods and Cones

For low-light vision

Rods: "Scotopic vision"

Sensitive to light energy

For high-light vision

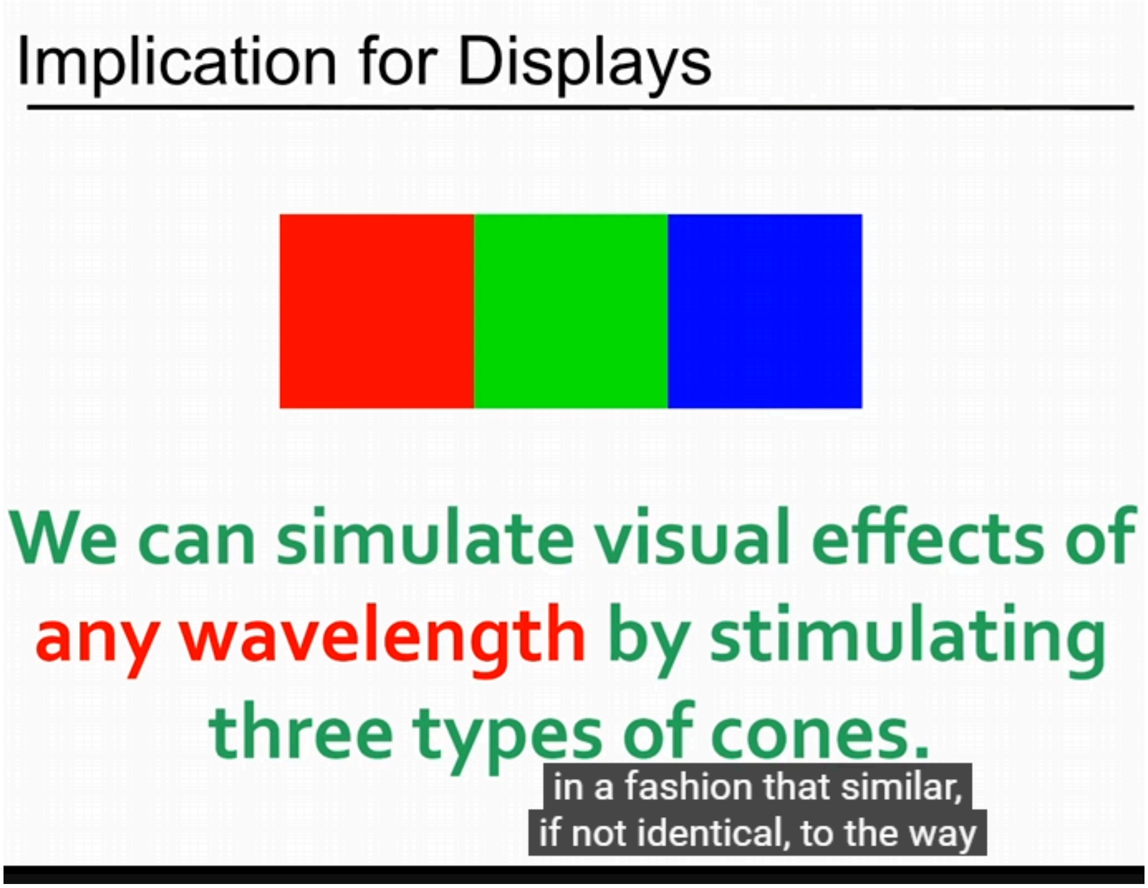
Cones: "Photopic vision"

Sensitive to color

but just the presence or absence
of something in front of you.

http://hamwaves.com/antennas/diel-rod/rods_cones.jpg

- ==Color blindness and metamers
  - Implication for Displays



Implication for Displays

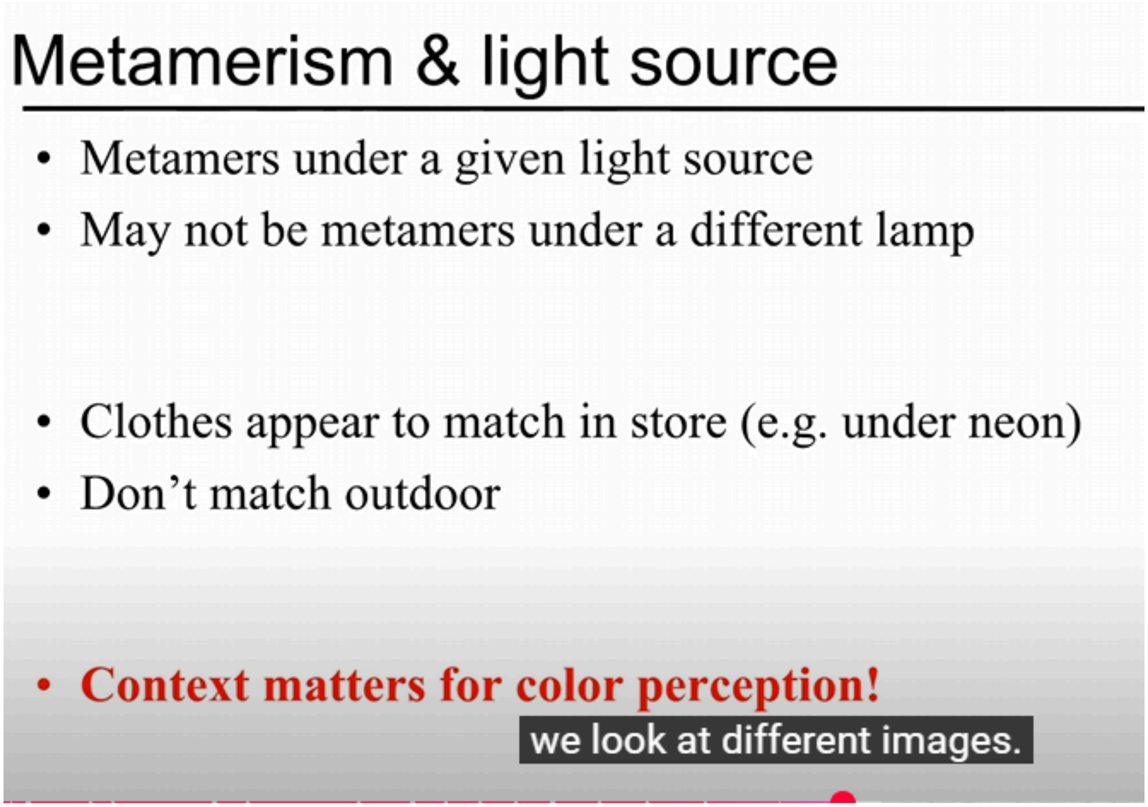
🟥🟩🟦

We can simulate visual effects of any wavelength by stimulating three types of cones.

in a fashion that similar,
if not identical, to the way

  - Long, Medium, Short wavelength of cone

- Metamerism & Light source



Metamerism & light source

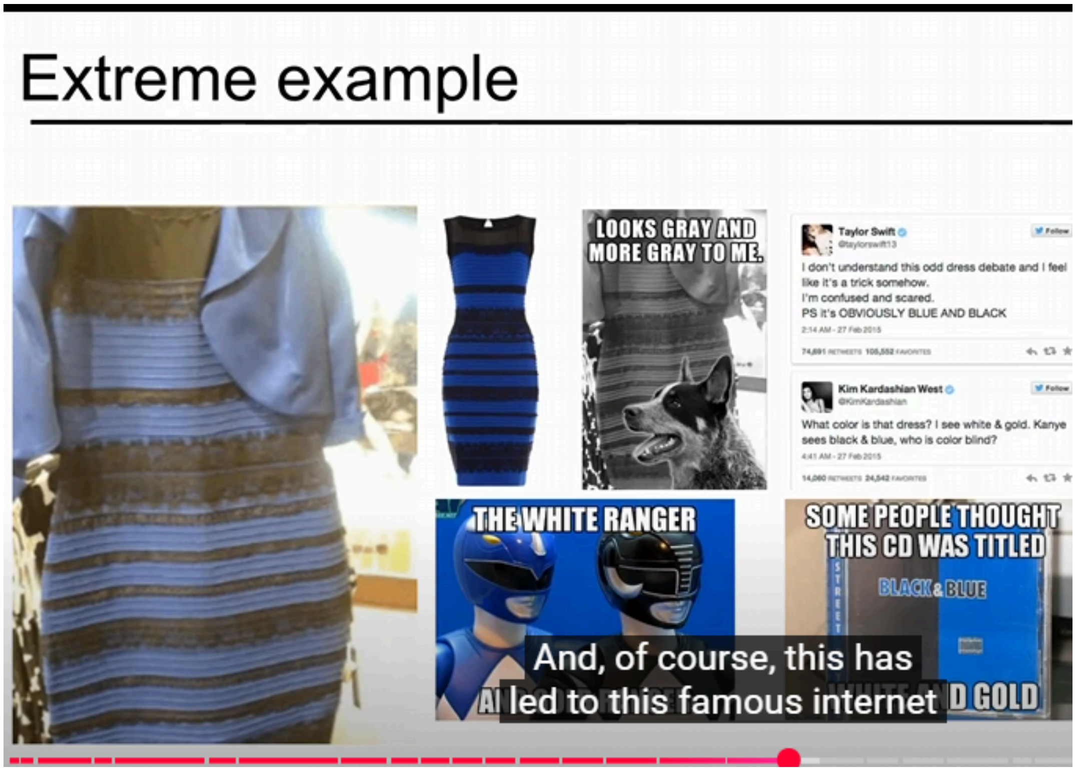
- Metamers under a given light source
- May not be metamers under a different lamp

- Clothes appear to match in store (e.g. under neon)
- Don’t match outdoor

- **Context matters for color perception!**

we look at different images.

  - Context matter, Example



Extreme example

LOOKS GRAY AND MORE GRAY TO ME.
THE WHITE RANGER
SOME PEOPLE THOUGHT THIS CD WAS TITLED
BLACK & BLUE
D GOLD
Follow
Follow

Taylor Swift @taylorswift13

I don't understand this odd dress debate and I feel like it's a trick somehow.
I'm confused and scared.
PS it's OBVIOUSLY BLUE AND BLACK

2:14 AM - 27 Feb 2015

74,891 RETWEETS 105,552 FAVORITES

Kim Kardashian West @KimKardashian

What color is that dress? I see white & gold. Kanye sees black & blue, who is color blind?

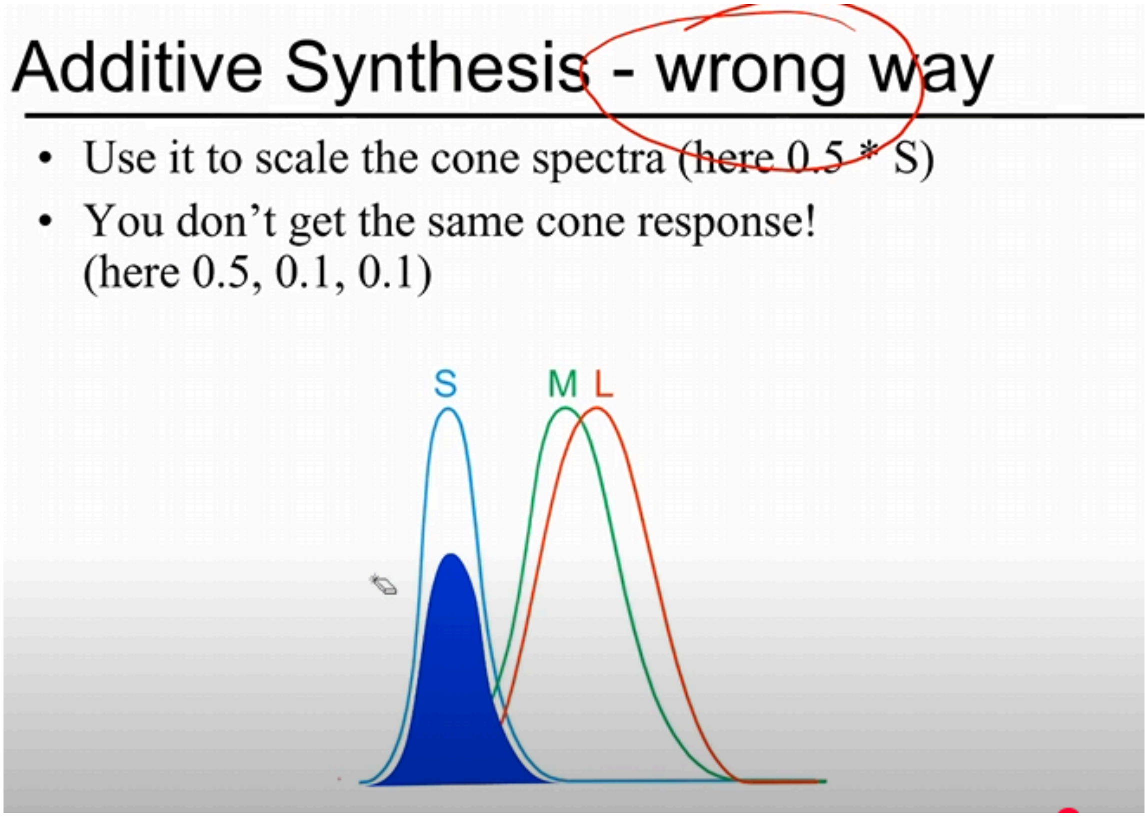
4:41 AM - 27 Feb 2015

14,060 RETWEETS 24,542 FAVORITES

And, of course, this has
led to this famous internet

- ==Color matching

- Wrong Way

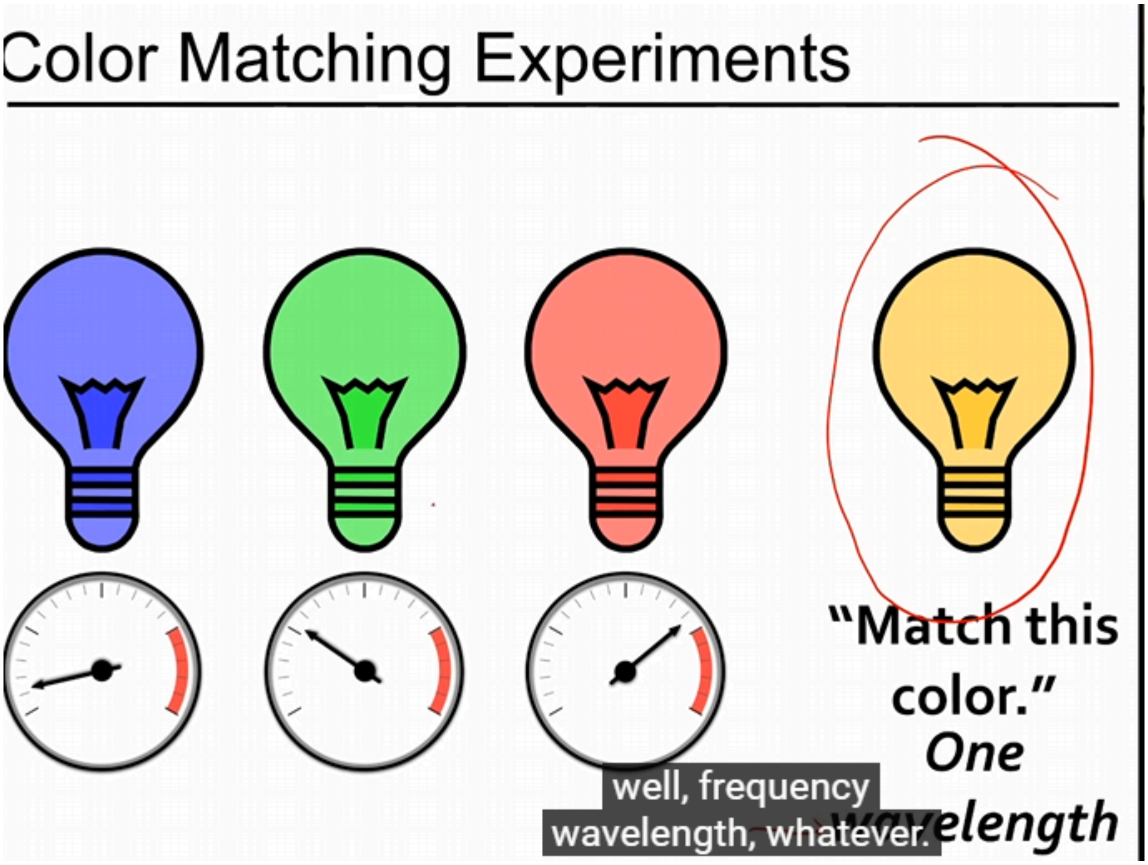


Additive Synthesis - wrong way

- Use it to scale the cone spectra (here 0.5 * S)
- You don't get the same cone response!
  (here 0.5, 0.1, 0.1)

  - They are not all independent (orthagonal), blue also have green and red cone

- Example

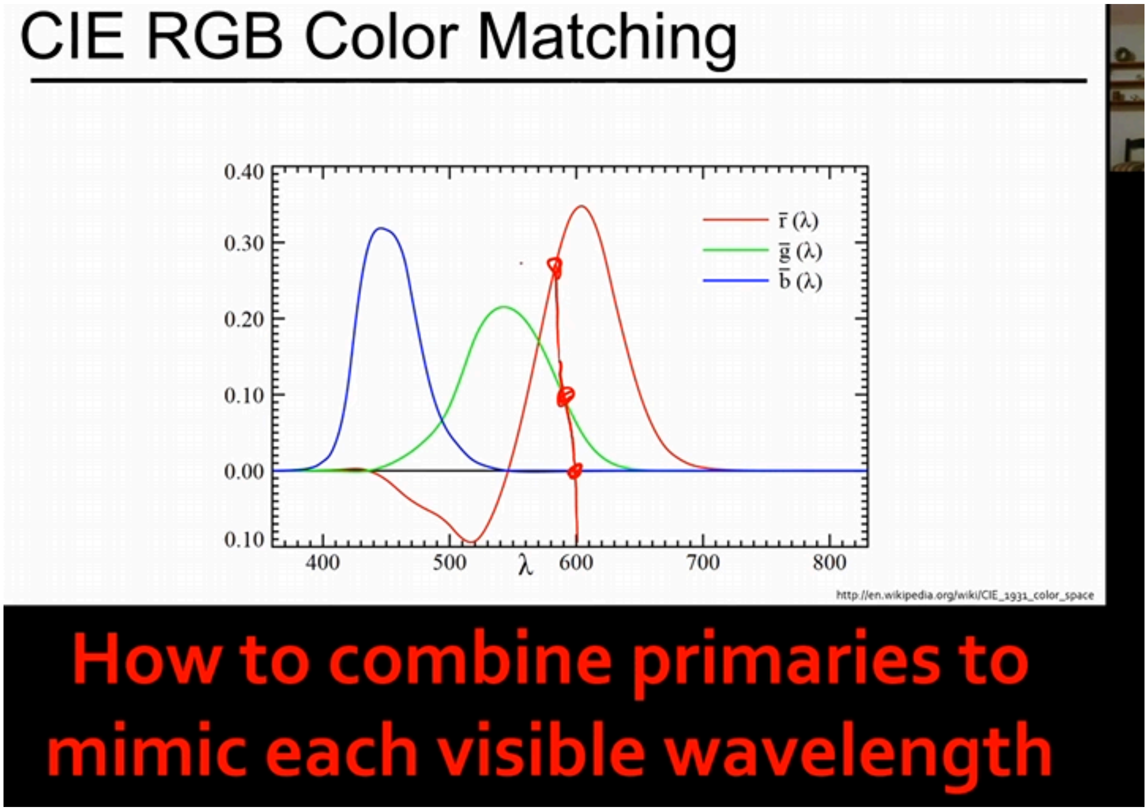


Color Matching Experiments

"Match this color."
One wavelength

well, frequency
wavelength, whatever.

    - CIE RGB Color Matching

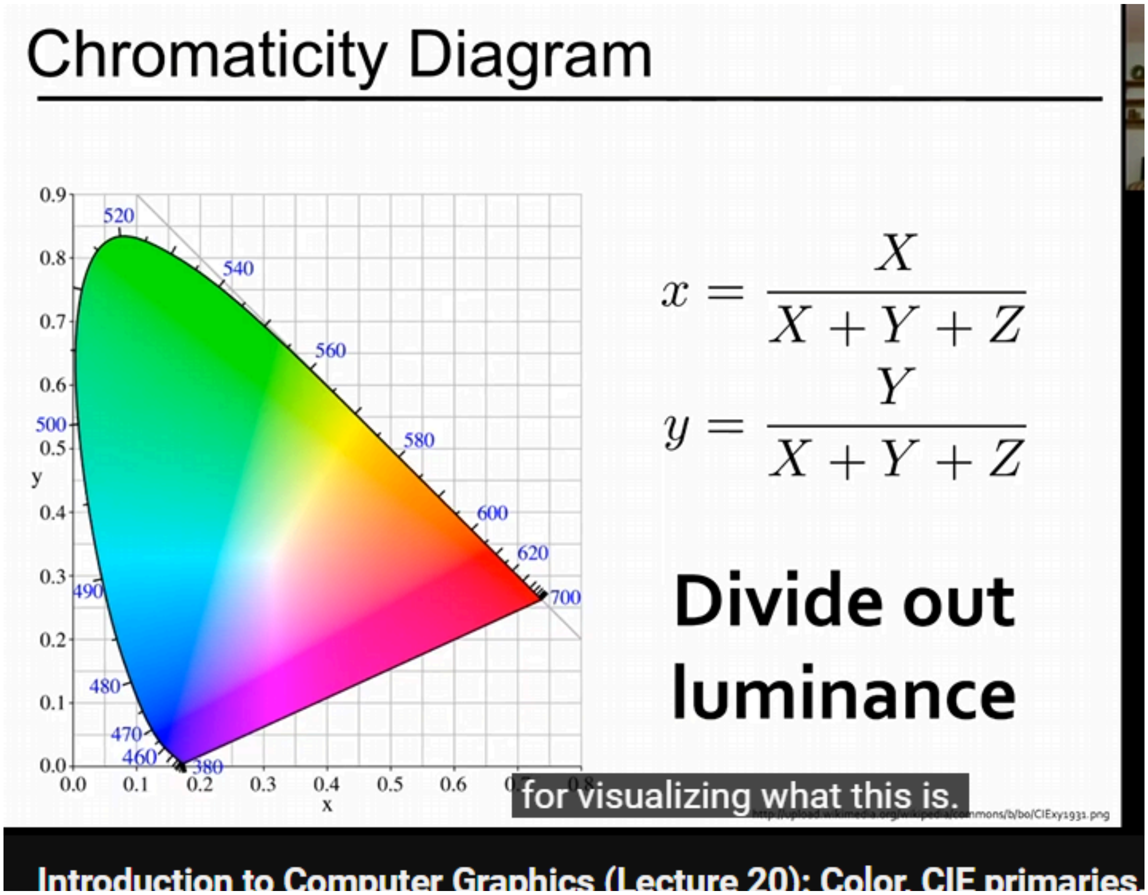


CIE RGB Color Matching

http://en.wikipedia.org/wiki/CIE_1931_color_space

How to combine primaries to mimic each visible wavelength

- ==Color spaces
    - Chromaticity Diagram (Full Color Space)



Chromaticity Diagram

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

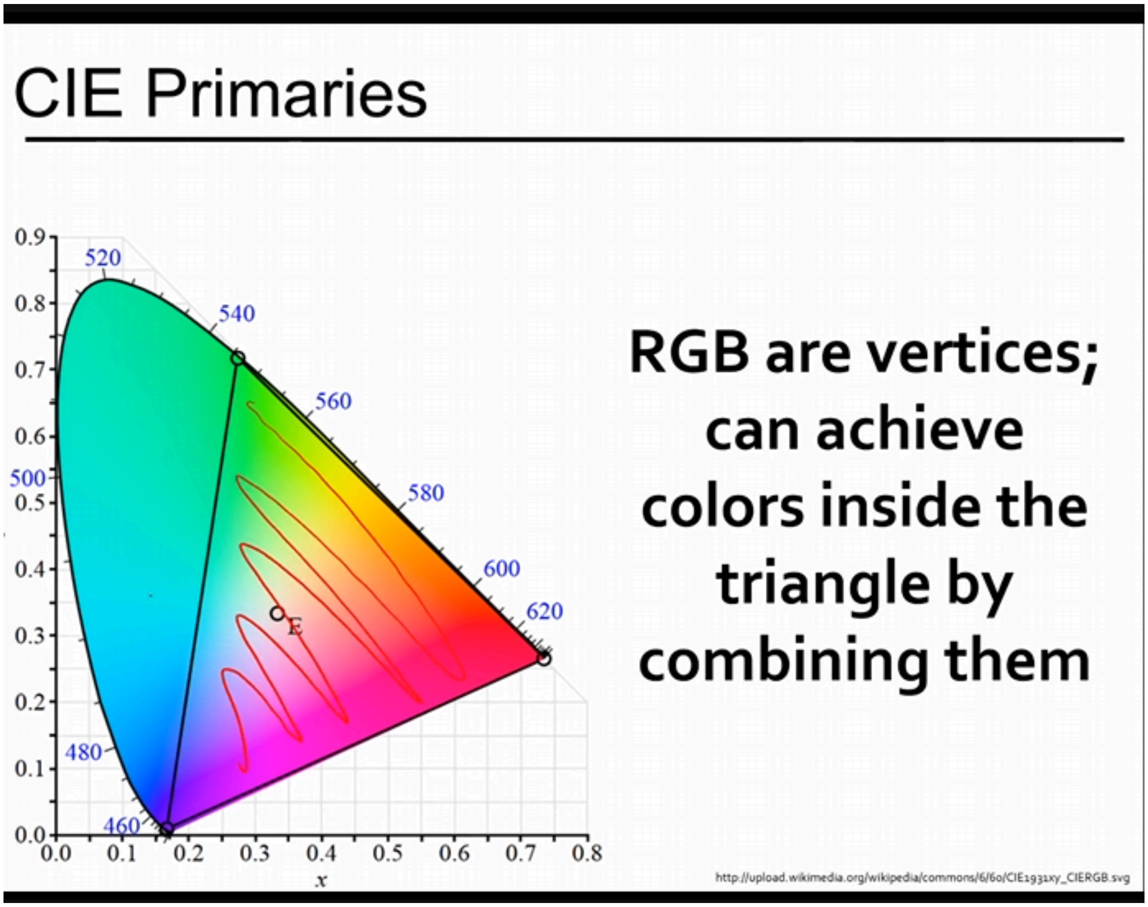
Divide out luminance

for visualizing what this is.

http://upload.wikimedia.org/wikipedia/commons/b/b0/CIExy1931.png

Introduction to Computer Graphics (Lecture 20): Color, CIE primaries

- CIE Primaries (triangle)

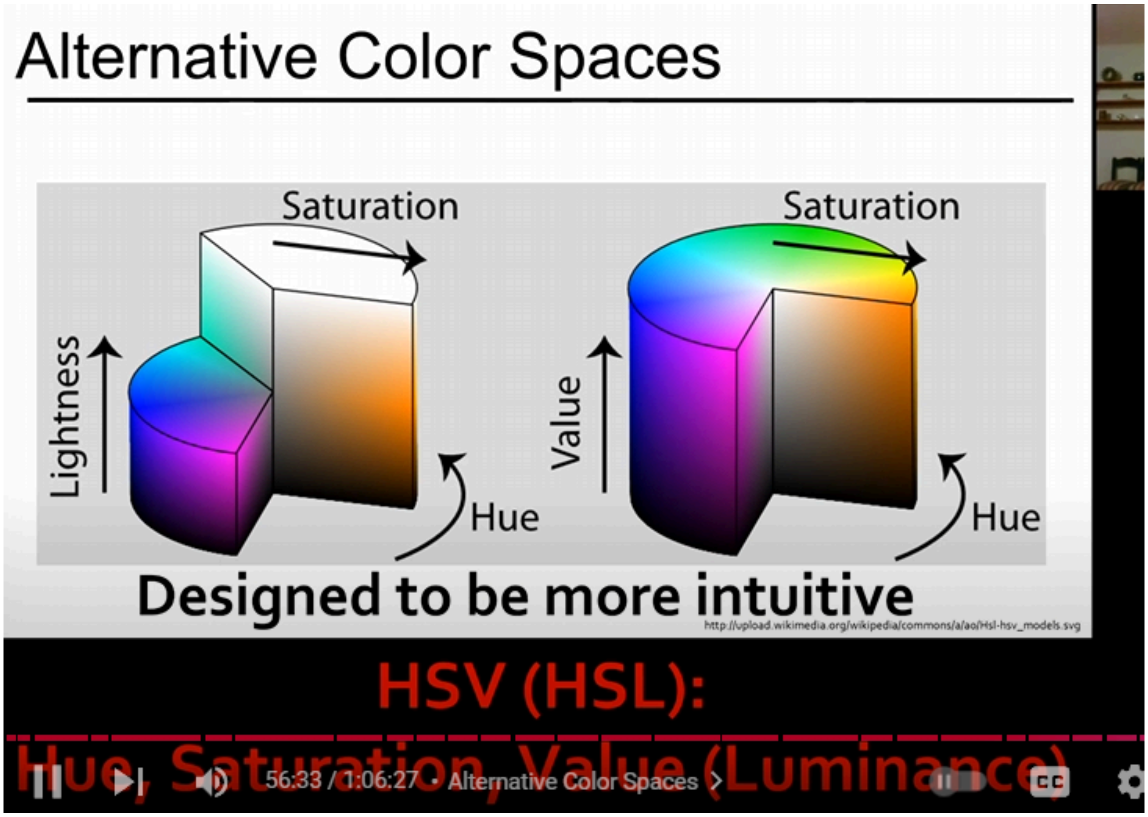


CIE Primaries

RGB are vertices; can achieve colors inside the triangle by combining them

http://upload.wikimedia.org/wikipedia/commons/6/60/CIE1931xy_CIERGB.svg

- HSV (Hue, Saturation, Value(Luminance))



Alternative Color Spaces

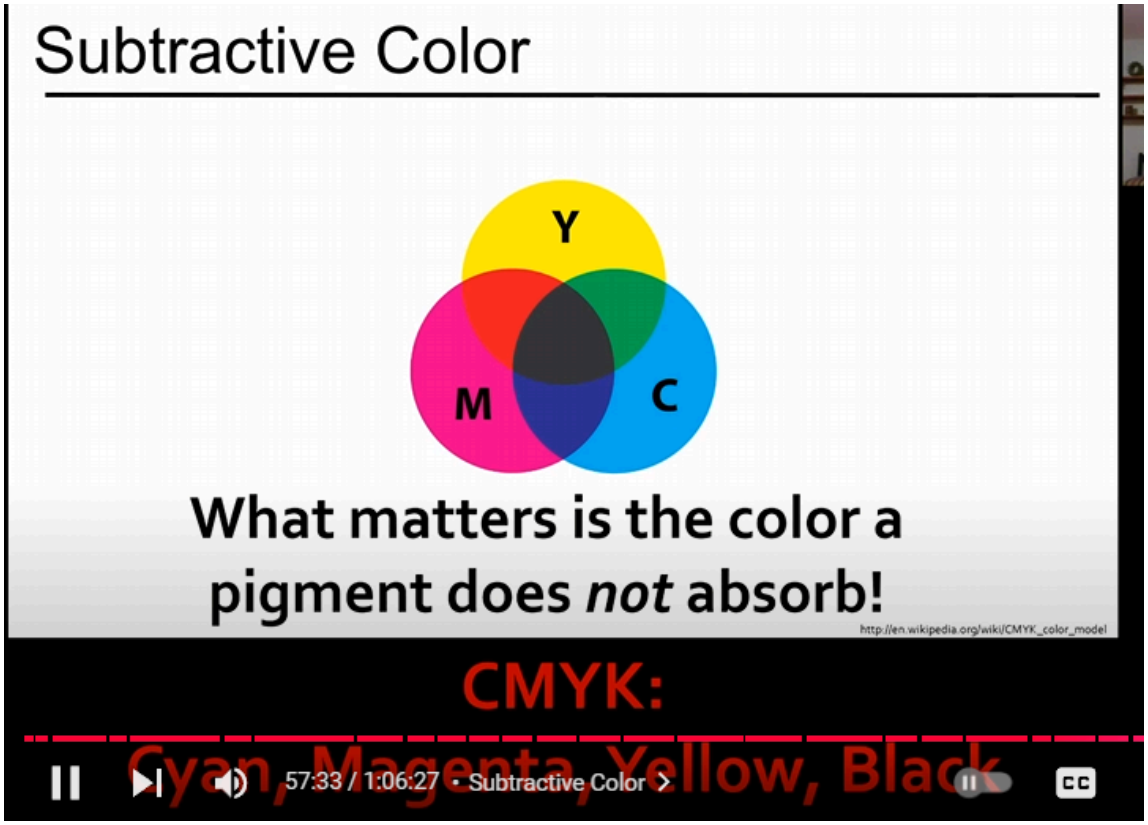
Designed to be more intuitive

http://upload.wikimedia.org/wikipedia/commons/a/a0/Hsl-hsv_models.svg

HSV (HSL):

Hue, Saturation, Value (Luminance)

- CMYK



Subtractive Color

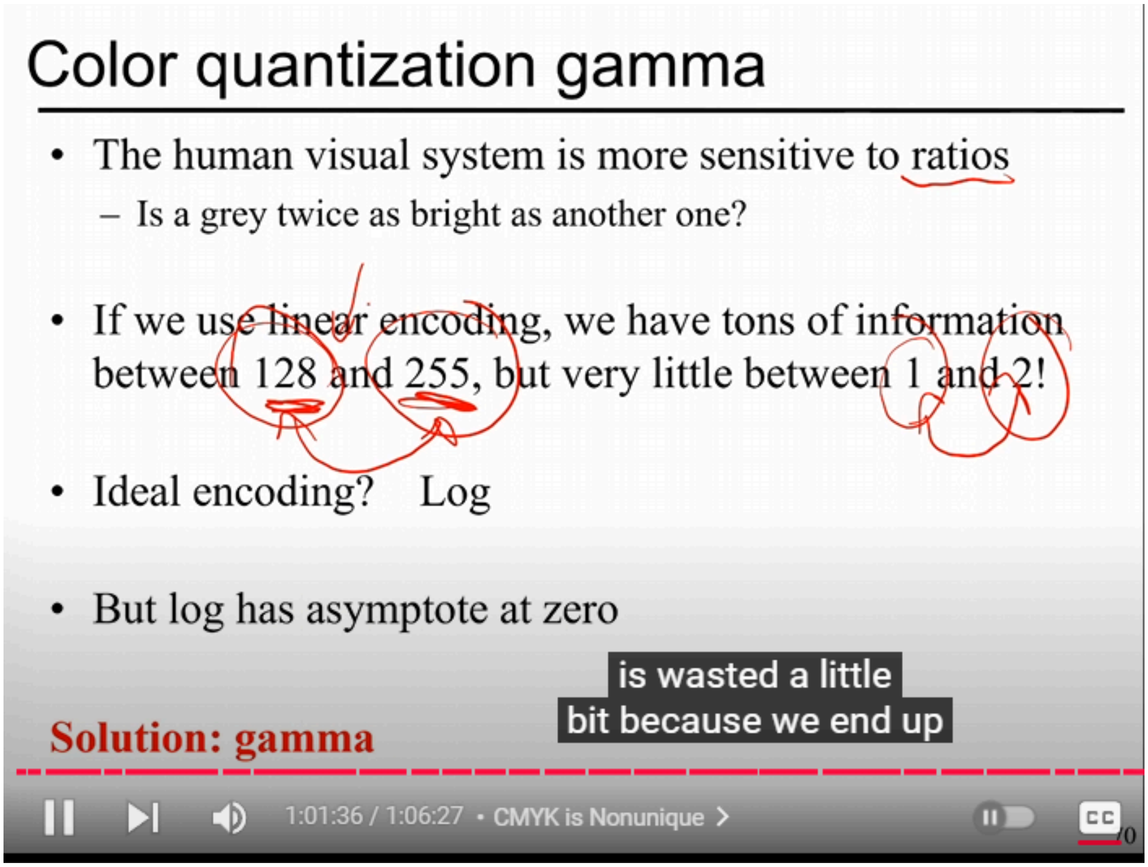
What matters is the color a pigment does *not* absorb!

http://en.wikipedia.org/wiki/CMYK_color_model

CMYK:
Cyan, Magenta, Yellow, Black

  - Subtract color from white

- Gamma
  - Color quantization gamma



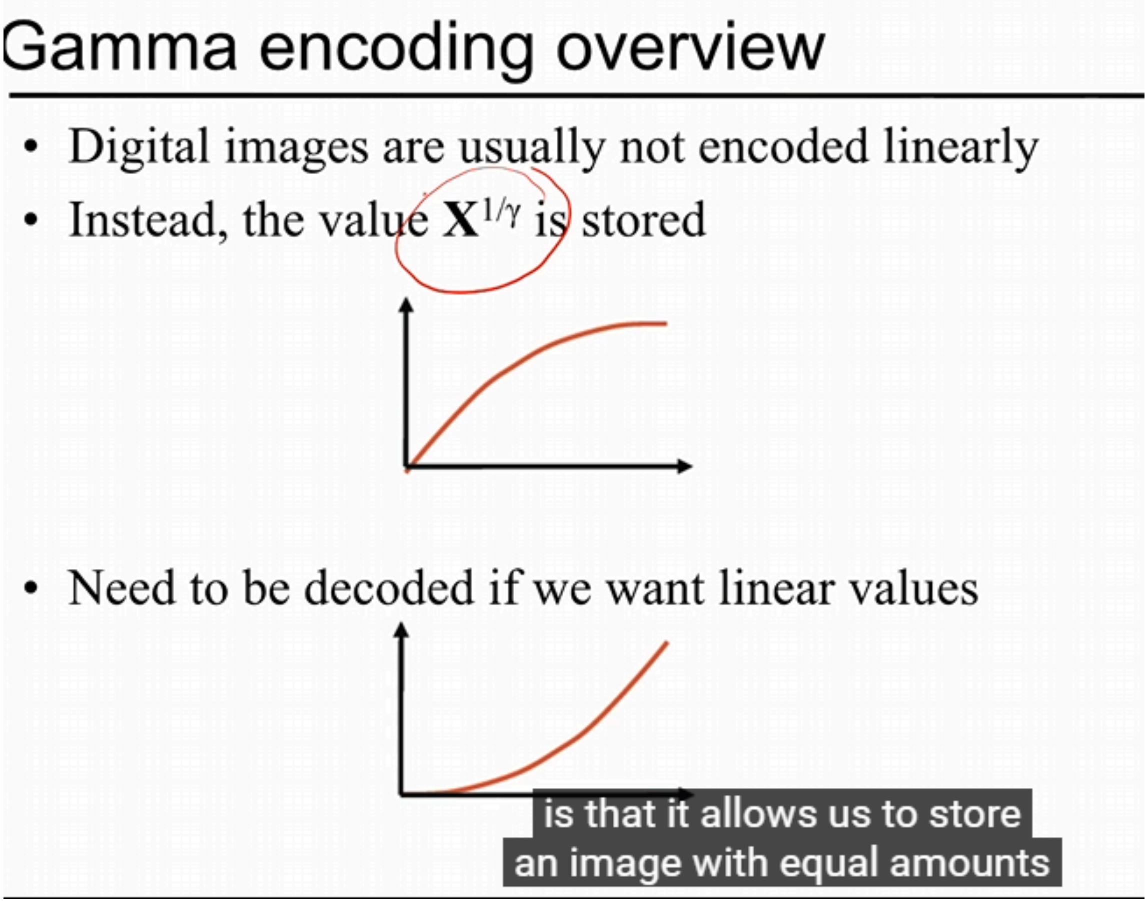
Color quantization gamma

- The human visual system is more sensitive to ratios
  - Is a grey twice as bright as another one?
- If we use linear encoding, we have tons of information between 128 and 255, but very little between 1 and 2!
- Ideal encoding? Log
- But log has asymptote at zero

is wasted a little
bit because we end up

Solution: gamma

- Gamma encoding

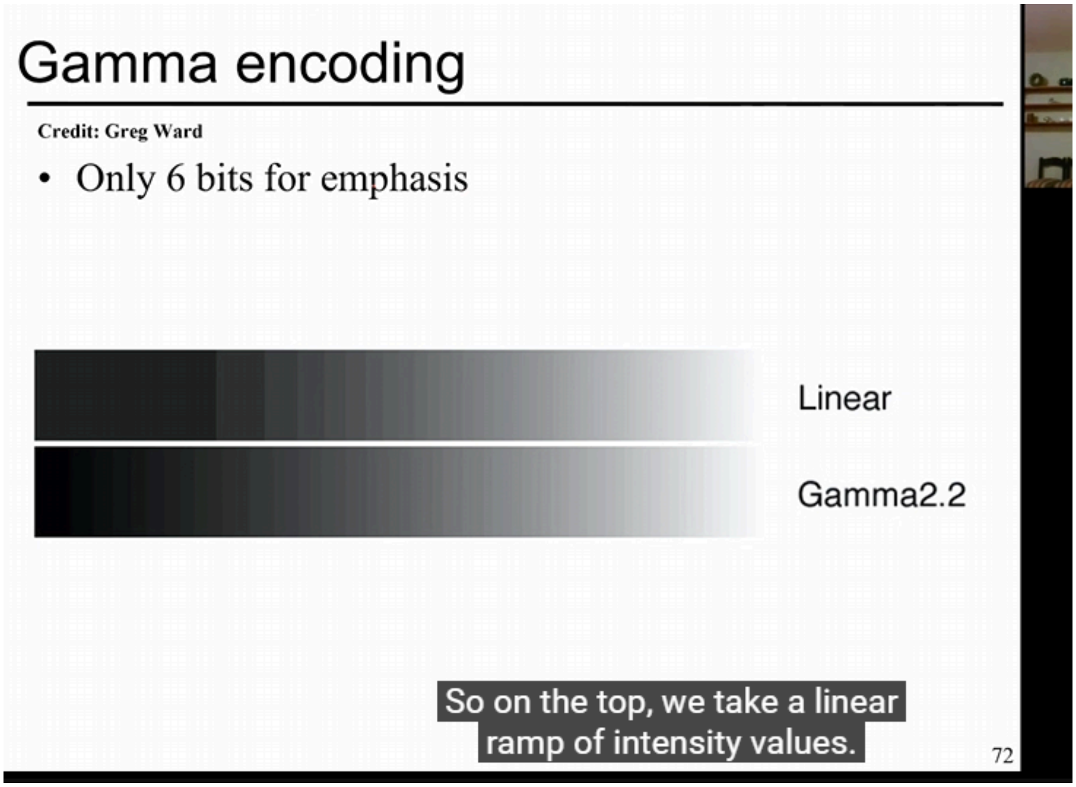


Gamma encoding overview

- Digital images are usually not encoded linearly
- Instead, the value $\mathbf{X}^{1/\gamma}$ is stored
- Need to be decoded if we want linear values

is that it allows us to store
an image with equal amounts

  - Example



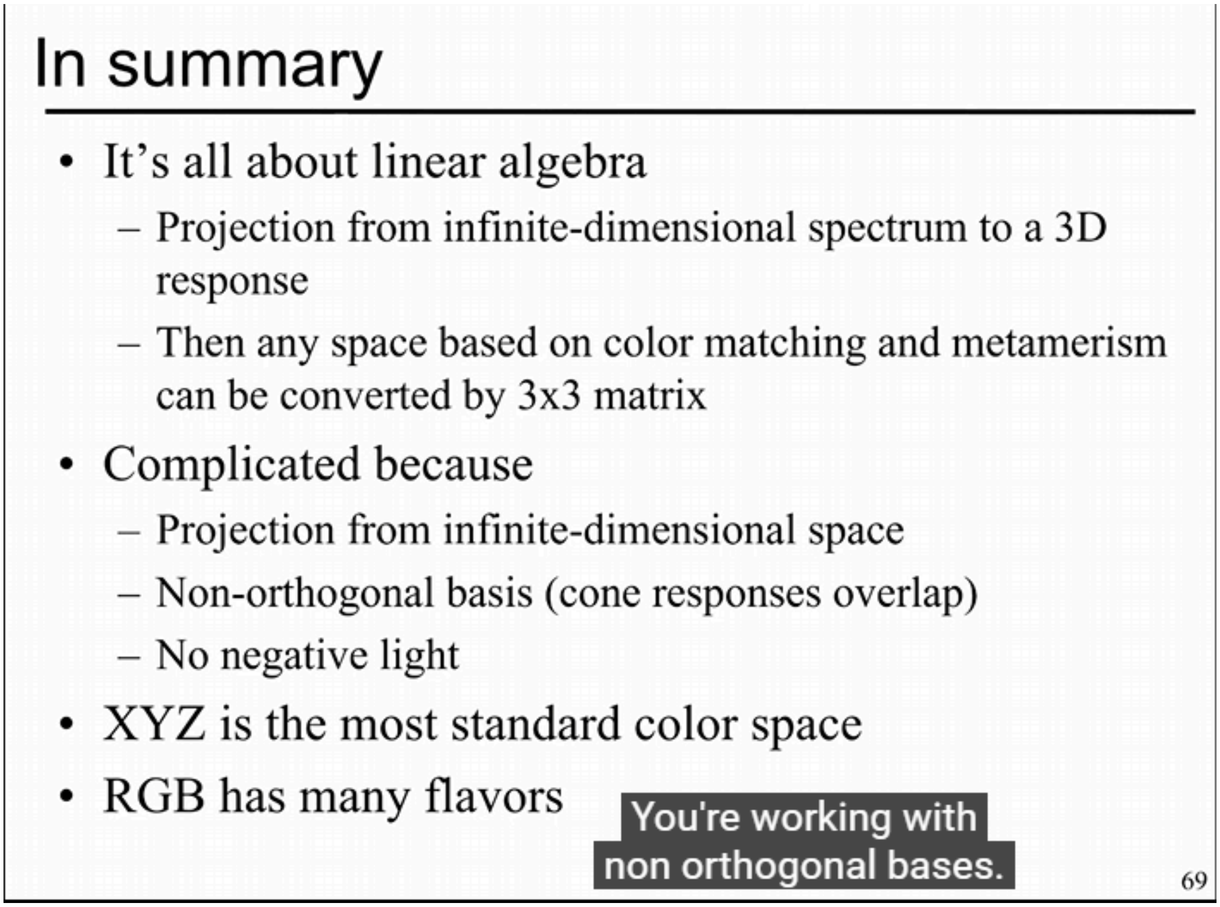
Gamma encoding

Credit: Greg Ward

- Only 6 bits for emphasis

So on the top, we take a linear
ramp of intensity values.

    - Summary



In summary

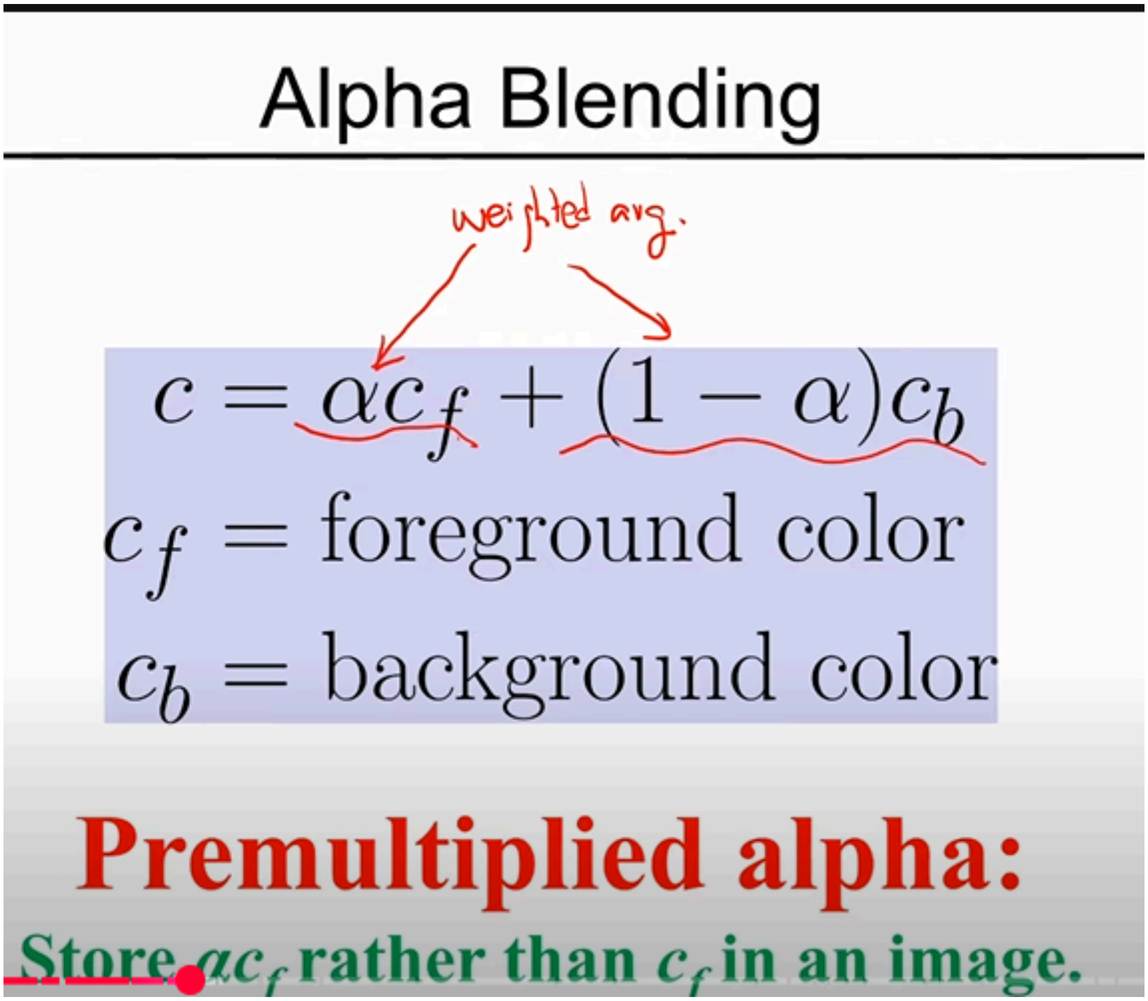
- It’s all about linear algebra
  - Projection from infinite-dimensional spectrum to a 3D response
  - Then any space based on color matching and metamerism can be converted by 3x3 matrix
- Complicated because
  - Projection from infinite-dimensional space
  - Non-orthogonal basis (cone responses overlap)
  - No negative light
- XYZ is the most standard color space
- RGB has many flavors

You're working with non orthogonal bases.

69

- **L21: Image Processing** (Post processing)
    - Basic Concept
        - Image processing can touch up images after rendering
    - Lots of per pixel filters

- Alpha Blending



Alpha Blending

weighted avg.

$$c = \underline{\alpha c_f} + \underline{(1 - \alpha)c_b}$$

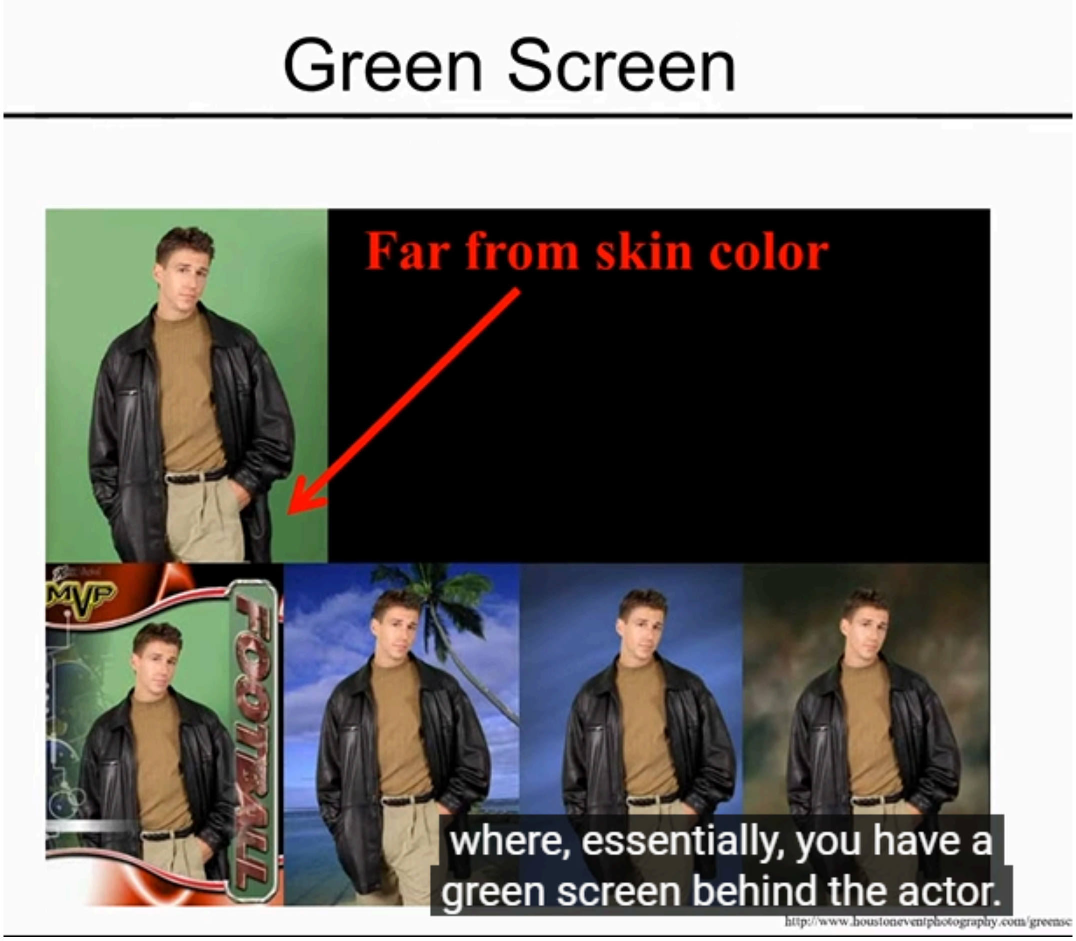
$$c_f = \text{foreground color}$$

$$c_b = \text{background color}$$

Premultiplied alpha:

Store αc_f rather than c_f in an image.

- Green Screen

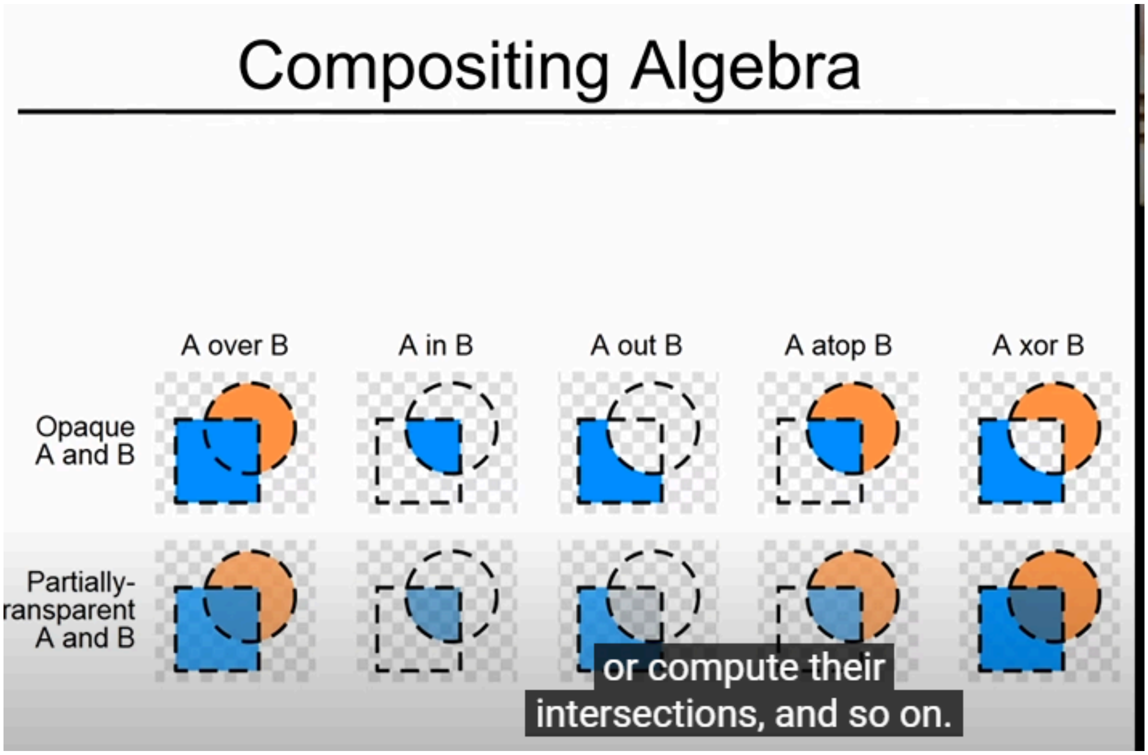


Green Screen

where, essentially, you have a green screen behind the actor.

http://www.houstoneventphotography.com/greensc

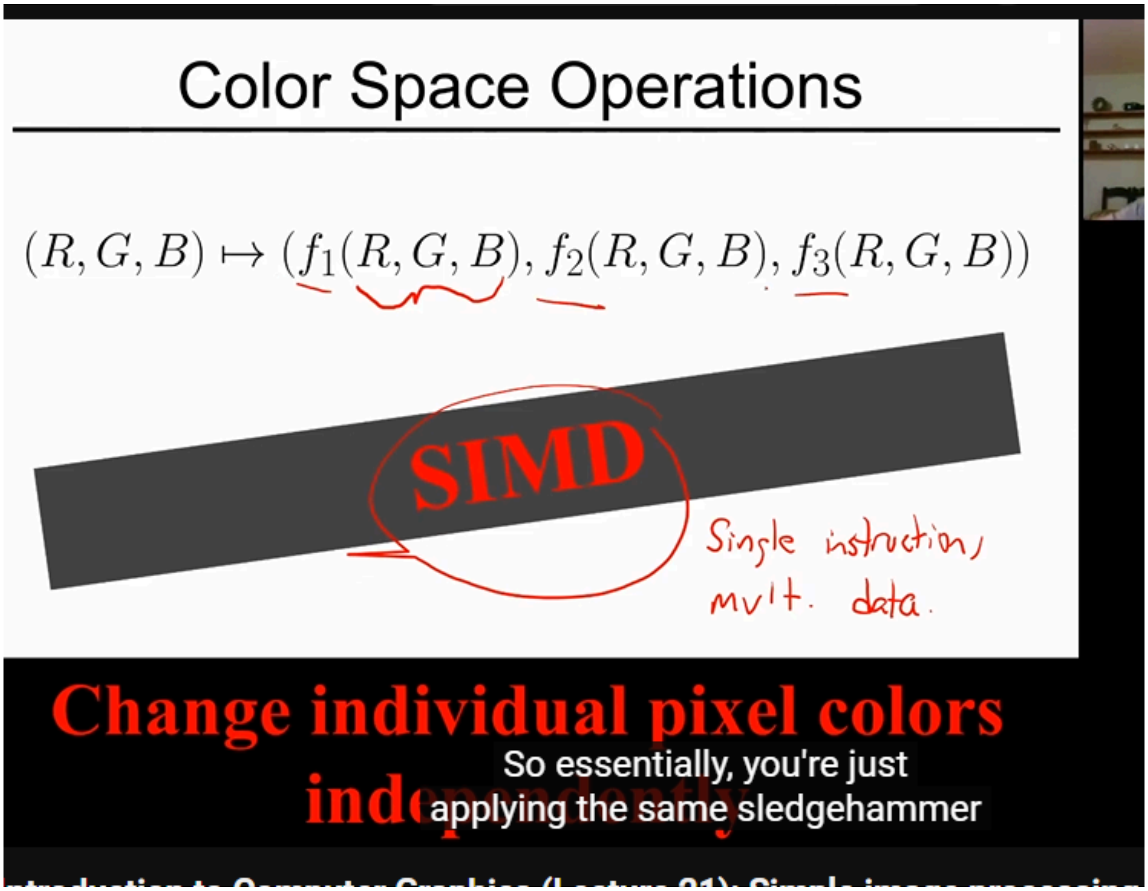
- Compositing Algebra



Compositing Algebra

or compute their intersections, and so on.

- Color Space Operations



Color Space Operations

$$(R,G,B) \mapsto (f_1(R,G,B), f_2(R,G,B), f_3(R,G,B))$$

SIMD

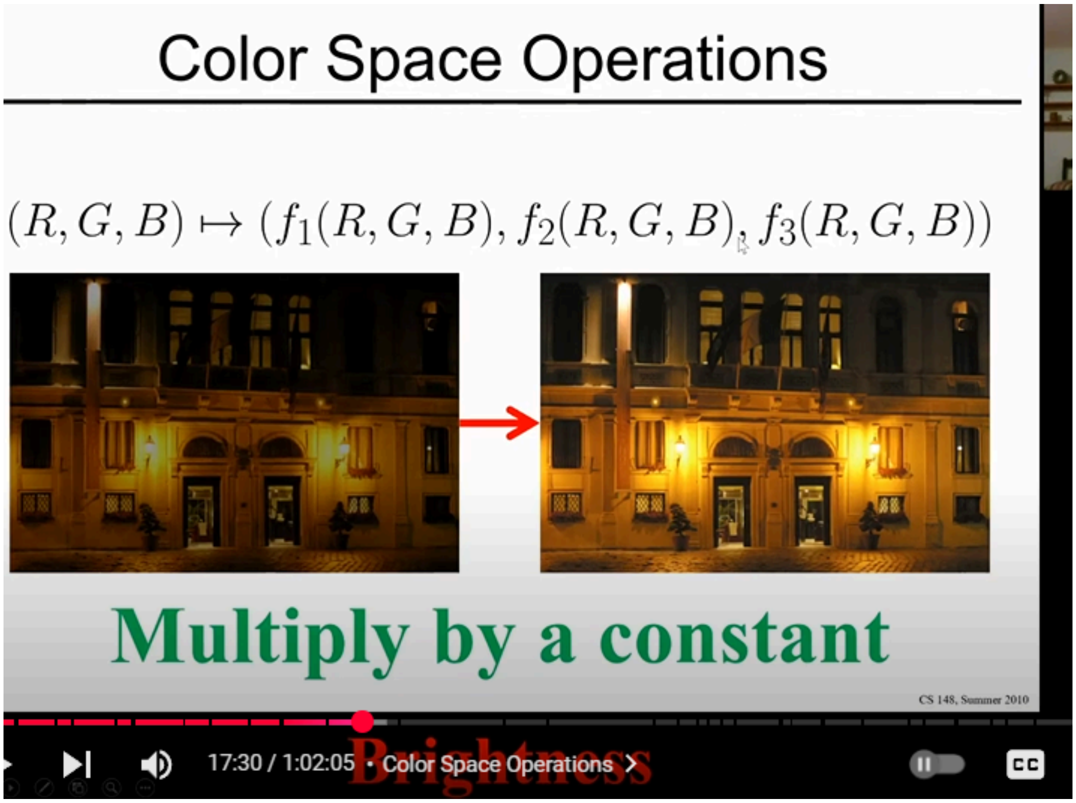
Single instruction, mult. data.

Change individual pixel colors indep

So essentially, you're just applying the same sledgehammer

Introduction to Computer Graphics (Lecture 21): Simple image processing

  - Apply every pixel to a function
  - Brightness



Color Space Operations

$$(R,G,B) \mapsto (f_1(R,G,B), f_2(R,G,B), f_3(R,G,B))$$

Multiply by a constant

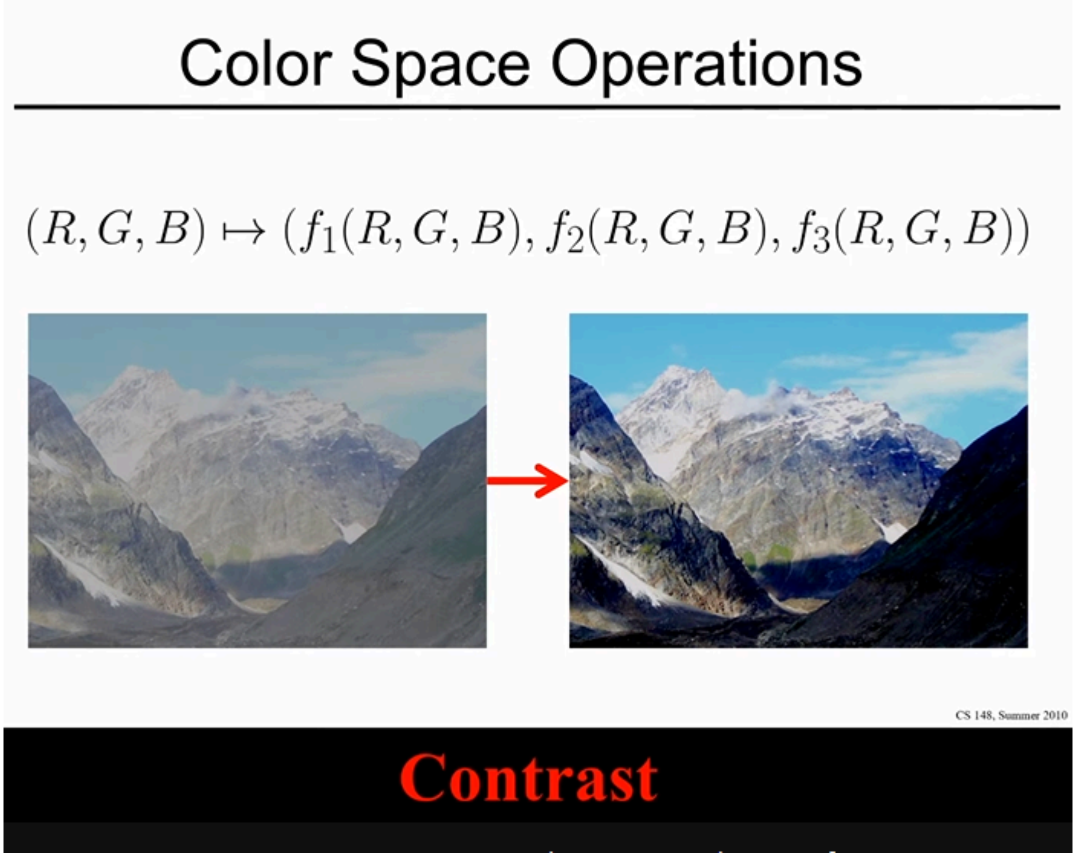
CS 148, Summer 2010

17:30 / 1:02:05 • Color Space Operations >

Brightness

    - Multiplay by a constant

- Contrast

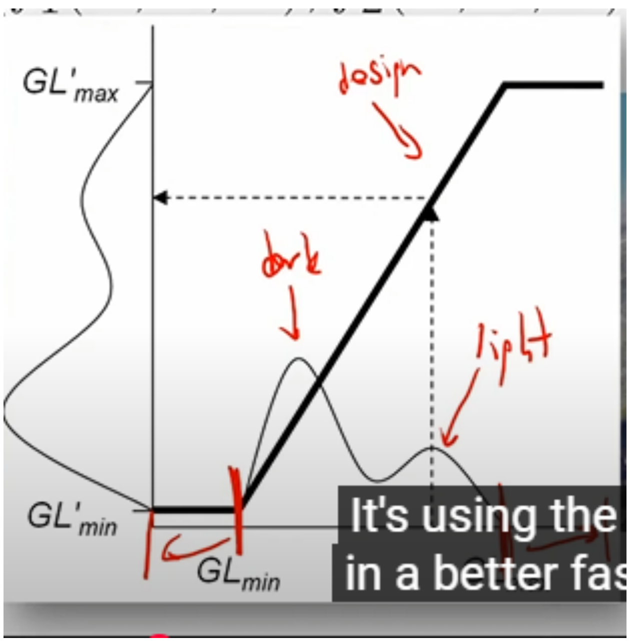


Color Space Operations

$$(R, G, B) \mapsto (f_1(R, G, B), f_2(R, G, B), f_3(R, G, B))$$

CS 148, Summer 2010

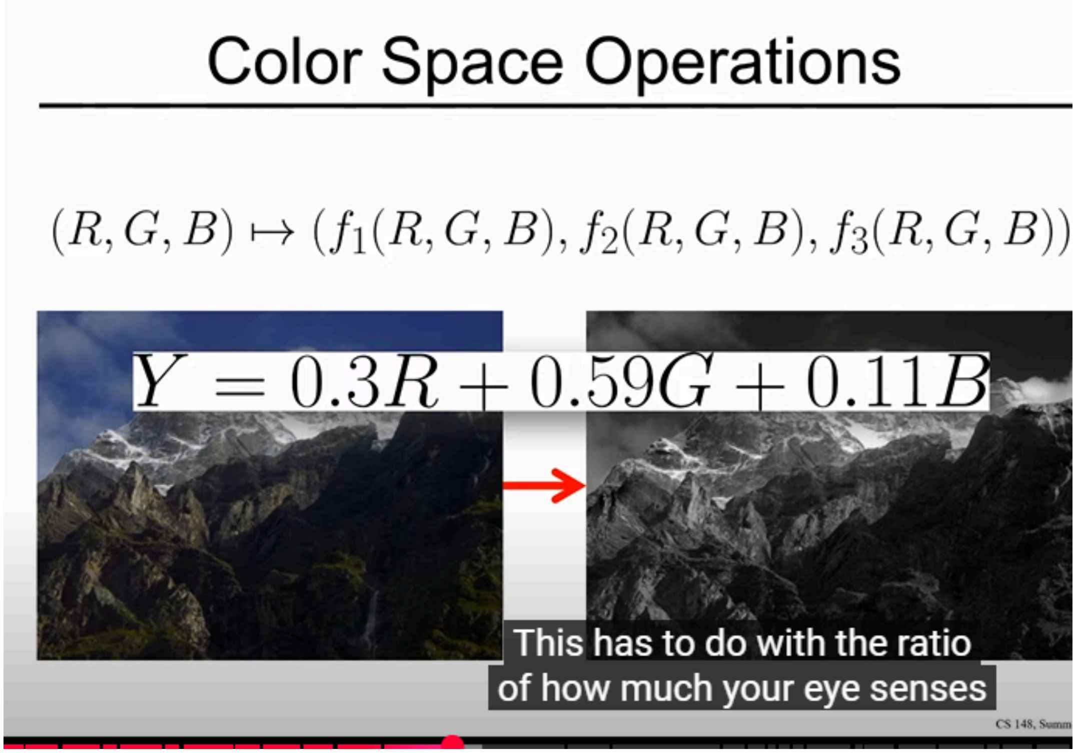
Contrast



It's using the
in a better fas

- Strengh the value to 0 - 1

- Desaturation



Color Space Operations

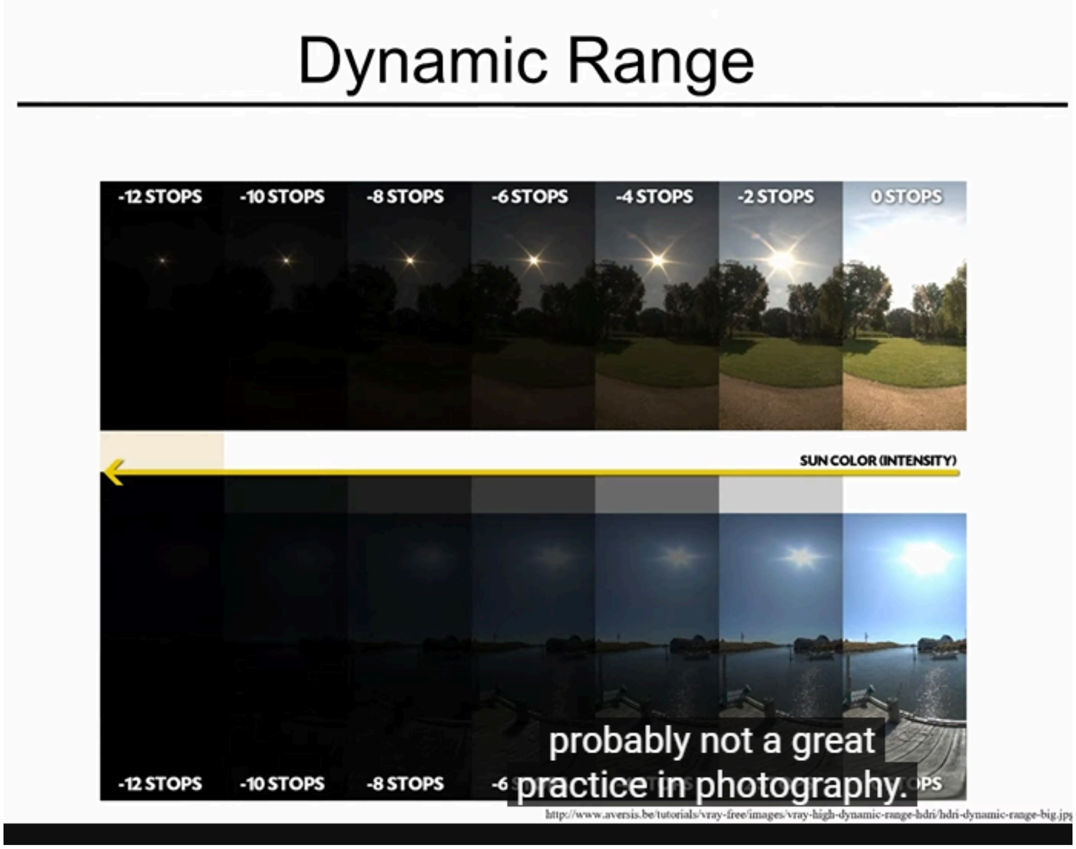
$$(R,G,B) \mapsto (f_1(R,G,B), f_2(R,G,B), f_3(R,G,B))$$

$$Y = 0.3R + 0.59G + 0.11B$$

This has to do with the ratio
of how much your eye senses

CS 148, Summ

- Dynamic Range (HDR)

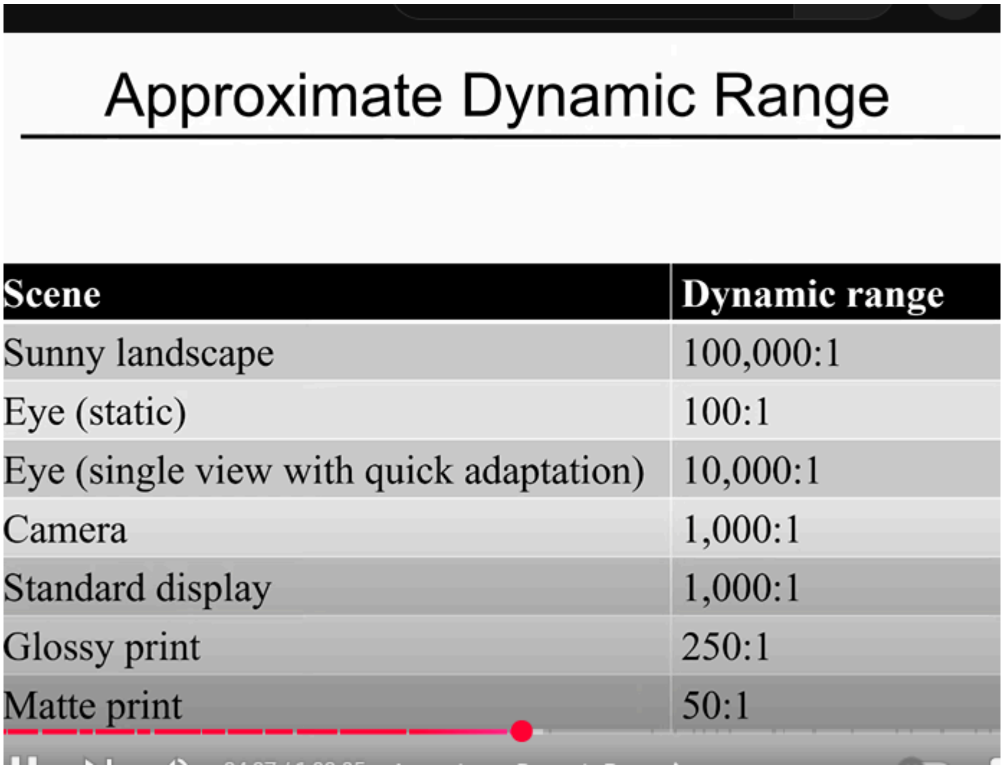


Dynamic Range

probably not a great
practice in photography.

http://www.aversis.be/tutorials/vray-free/images/vray-high-dynamic-range-hdri/hdri-dynamic-range-big.jpg

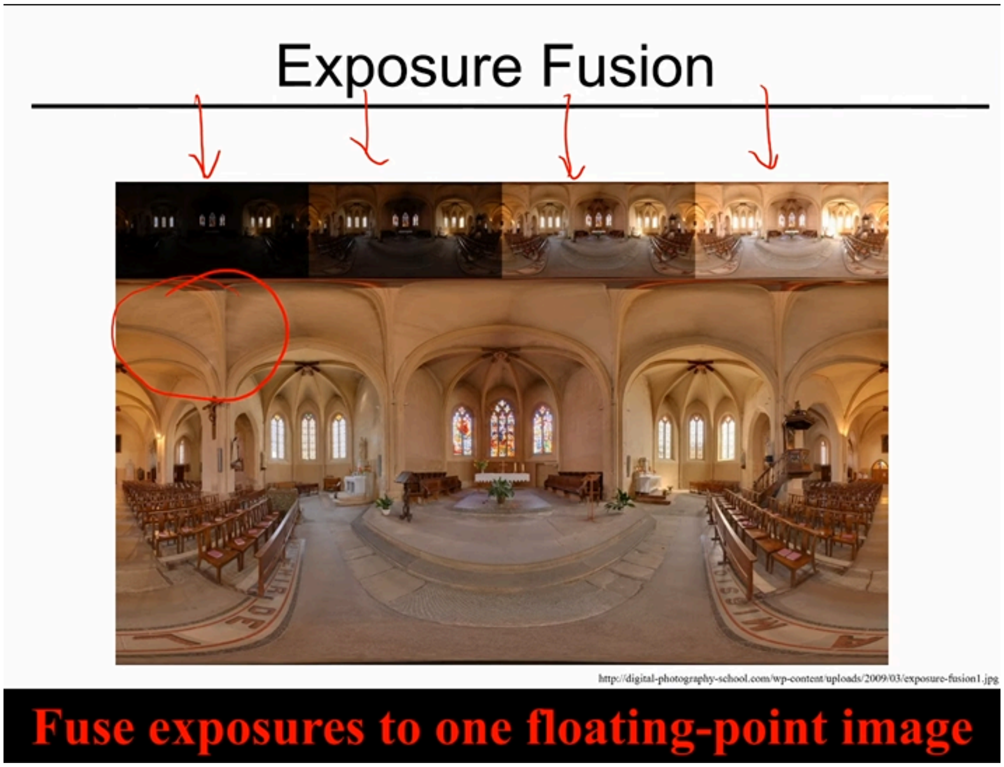
- Approximate Dynamic Rnage



Approximate Dynamic Range

Scene	Dynamic range
Sunny landscape	100,000:1
Eye (static)	100:1
Eye (single view with quick adaptation)	10,000:1
Camera	1,000:1
Standard display	1,000:1
Glossy print	250:1
Matte print	50:1

- Exposure Fusion

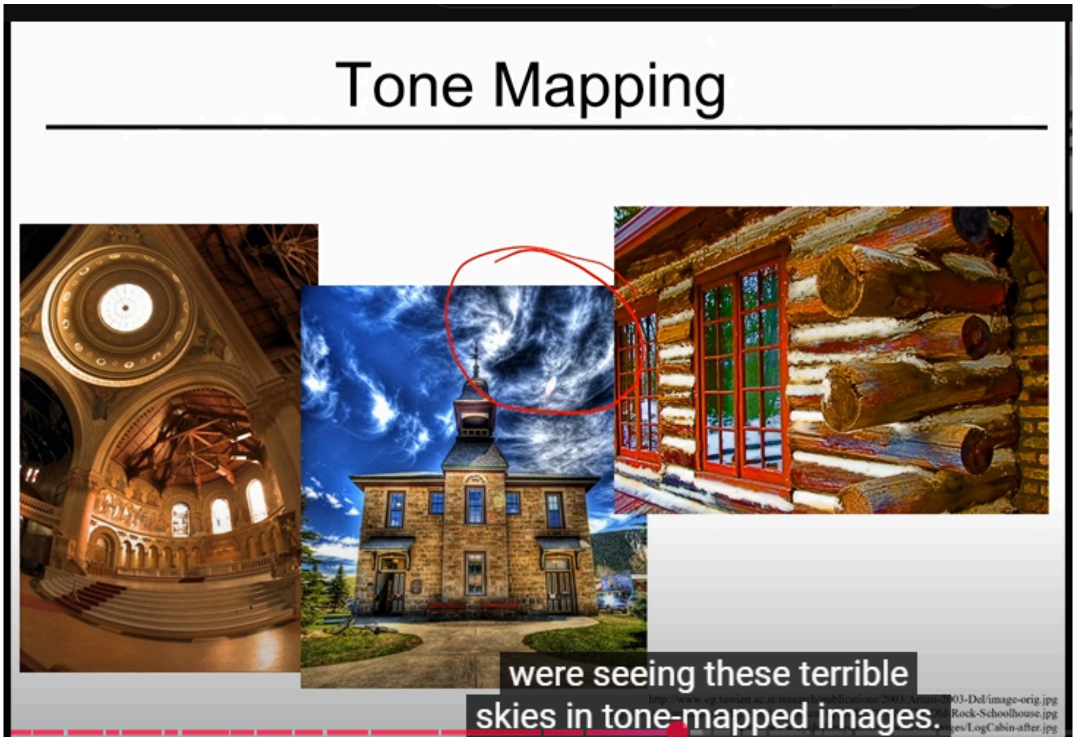


Exposure Fusion

http://digital-photography-school.com/wp-content/uploads/2009/03/exposure-fusion1.jpg

Fuse exposures to one floating-point image

- Tone Mapping



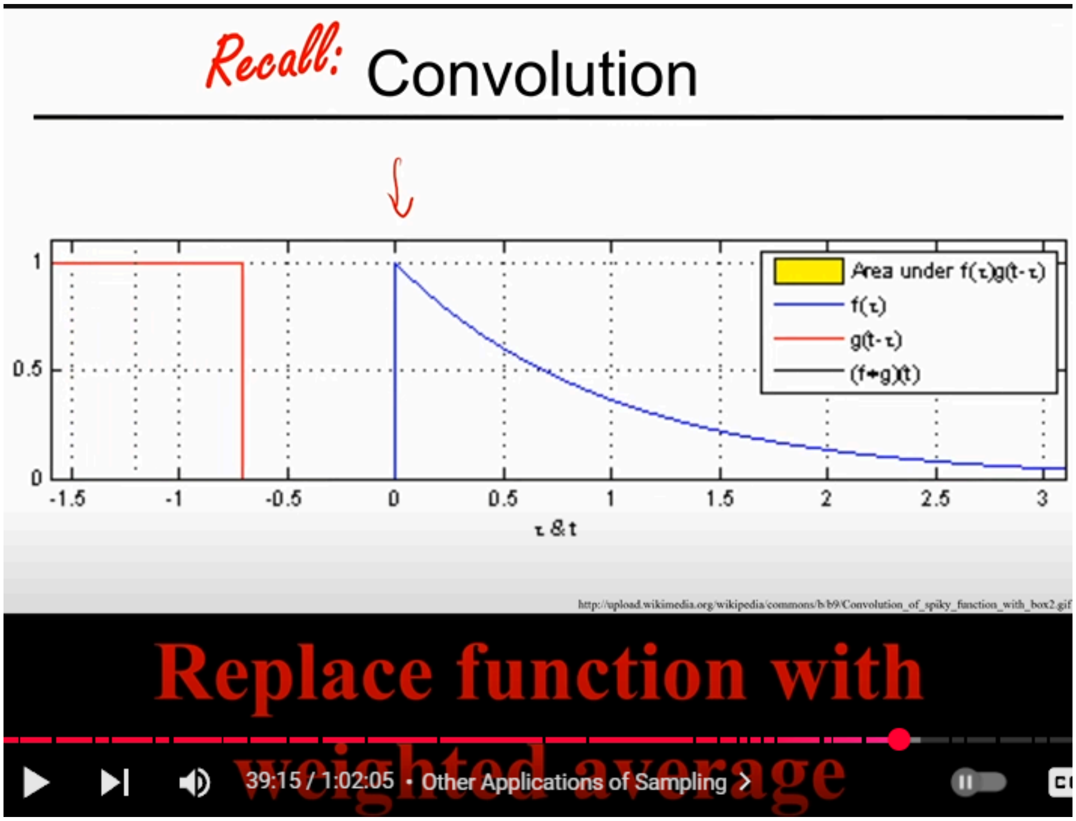
Tone Mapping

were seeing these terrible
skies in tone-mapped images.

http://www.cg.tuwien.ac.at/research/publications/2003/Artusi-2003-Del/image-orig.jpg
Old-Rock-Schoolhouse.jpg
ges/LogCabin-after.jpg

- Minification
  - Smaller image
- Magnification
  - Gigger image

- Filters involving larger neighborhoods, onlinearity
  - Convolution



Recall: Convolution

http://upload.wikimedia.org/wikipedia/commons/b/b9/Convolution_of_spiky_function_with_box2.gif

Replace function with
weighted average

- 3x3 3x3. calculate

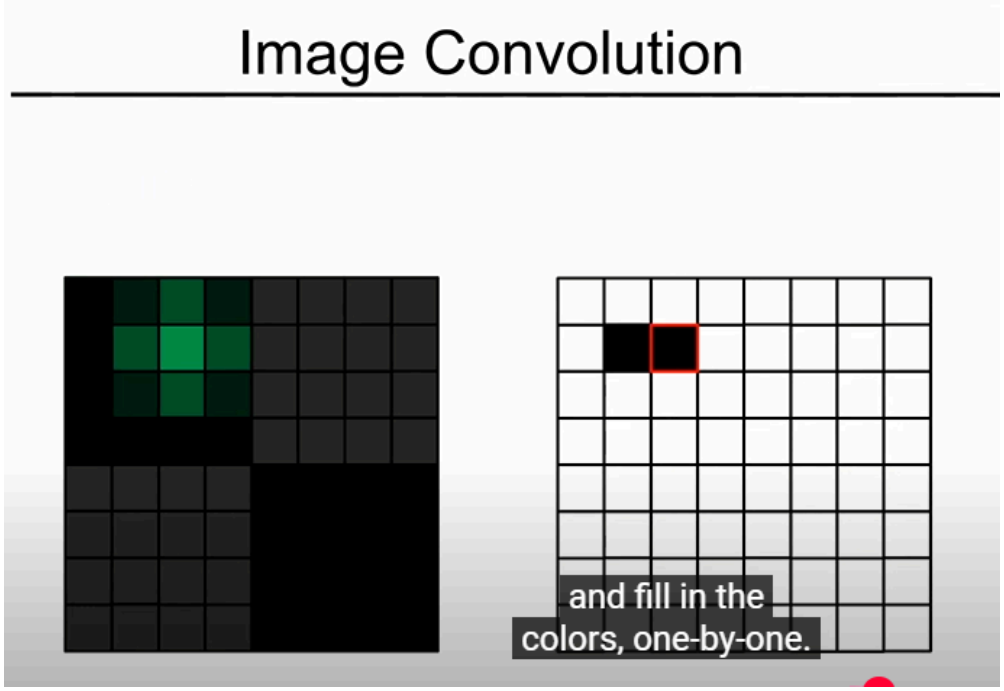
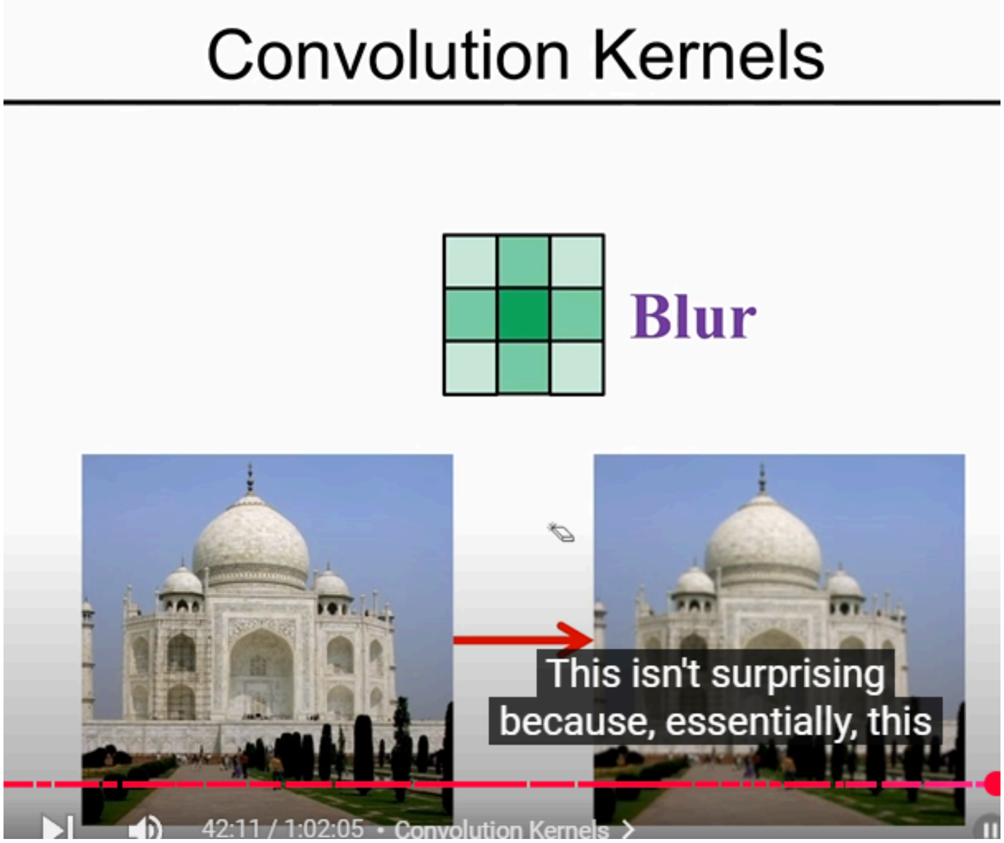


Image Convolution

and fill in the colors, one-by-one.

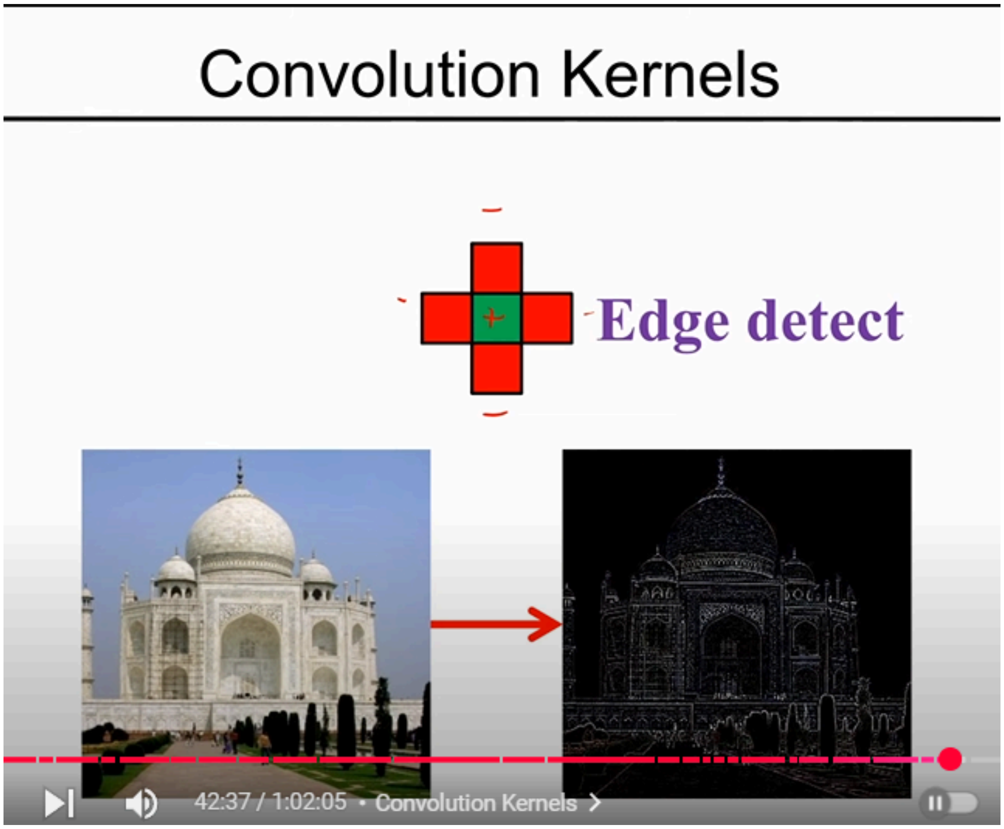
- Example: Blur



Convolution Kernels

This isn't surprising because, essentially, this

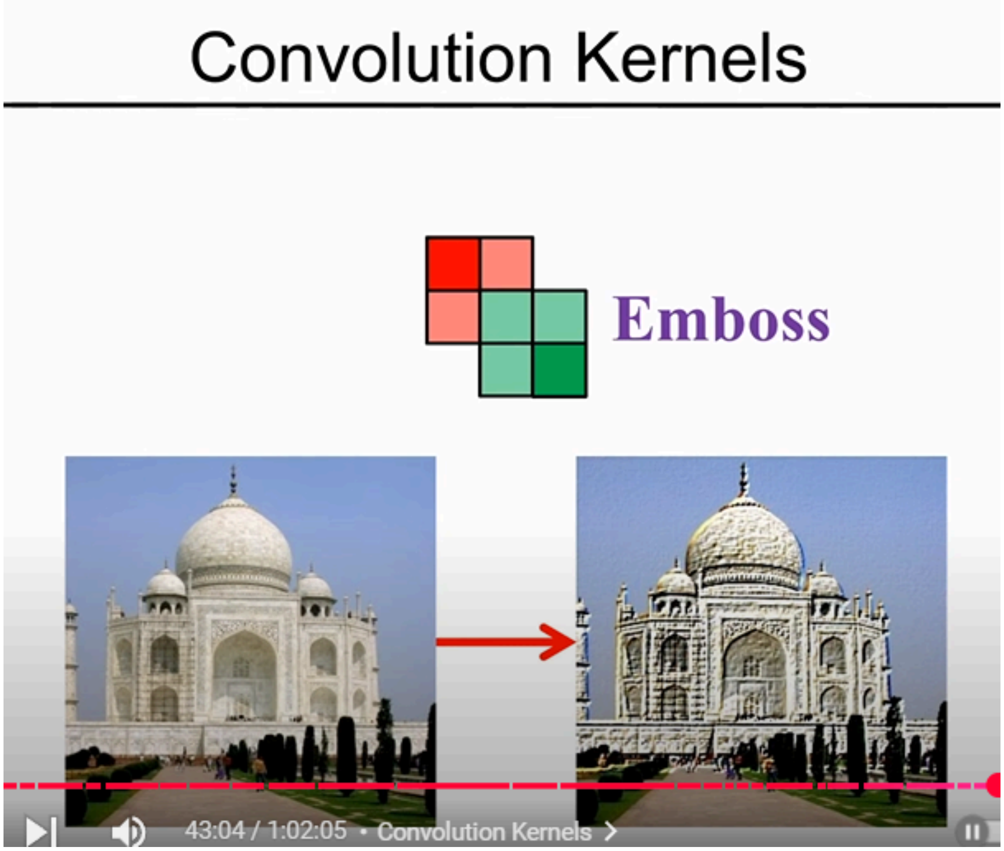
- Example: Edge detect



Convolution Kernels

Edge detect

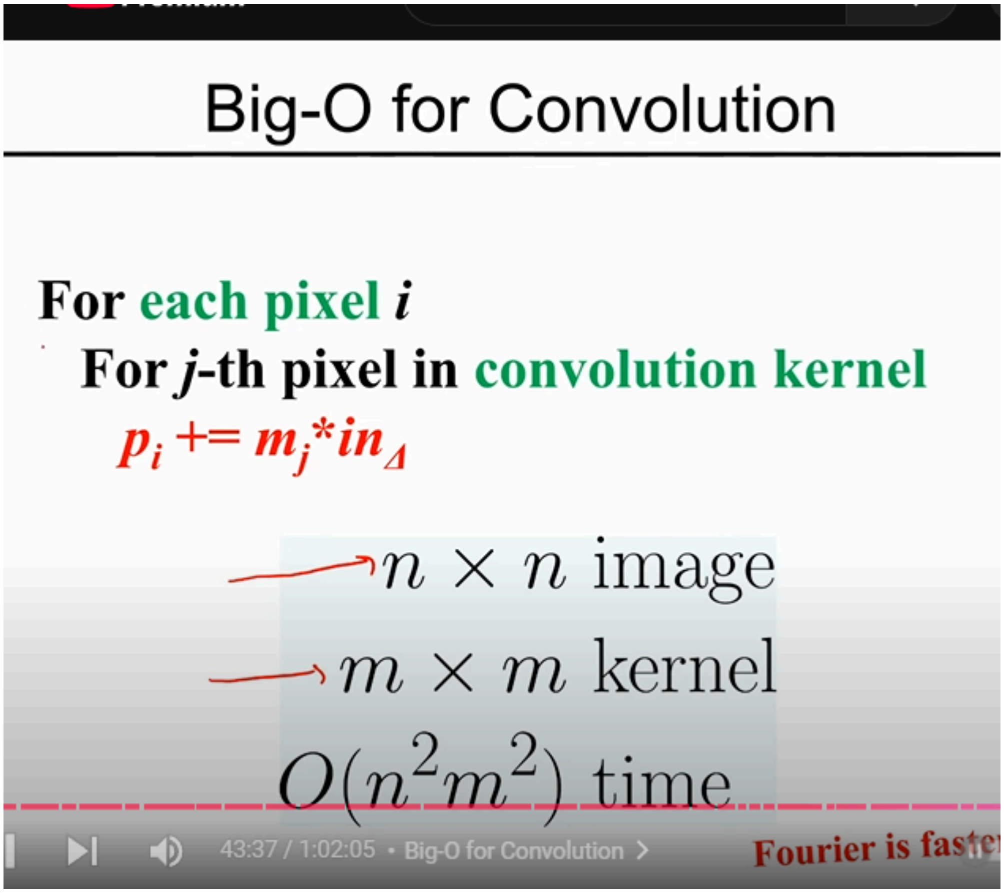
- Example: Emboss



Convolution Kernels

Emboss

    - Big-O for convolution



Big-O for Convolution

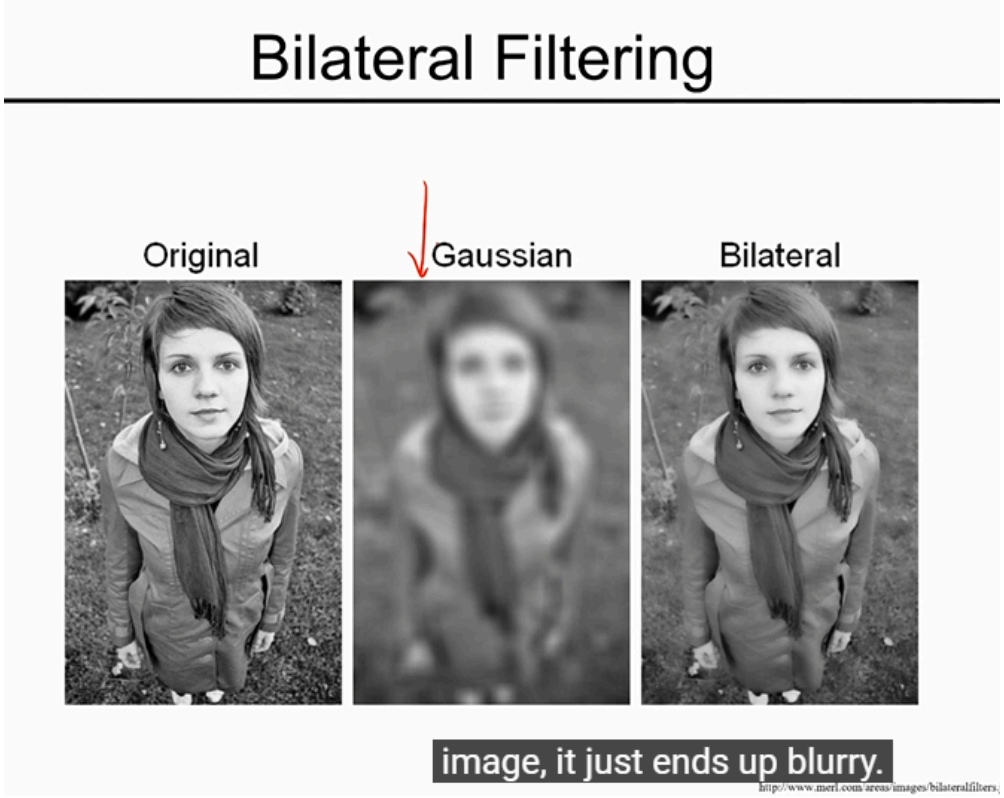
```
For each pixel i
    For j-th pixel in convolution kernel
        $p_i += m_j * in_\Delta$
```

$n \times n$ image

$m \times m$ kernel

$O(n^2 m^2)$ time

- Edge-perserving filtering
    - Unsharp Mask
    - Bilateral Filtering

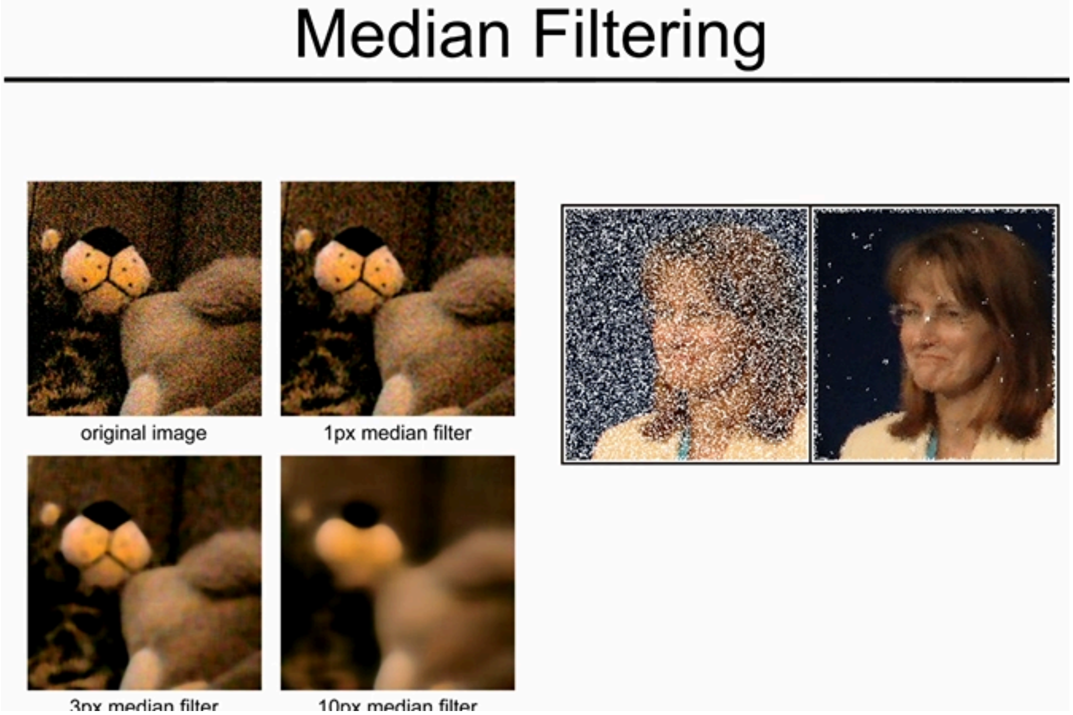


Bilateral Filtering

image, it just ends up blurry.

http://www.merl.com/areas/images/bilateralfilters

- Median filtering



Median Filtering

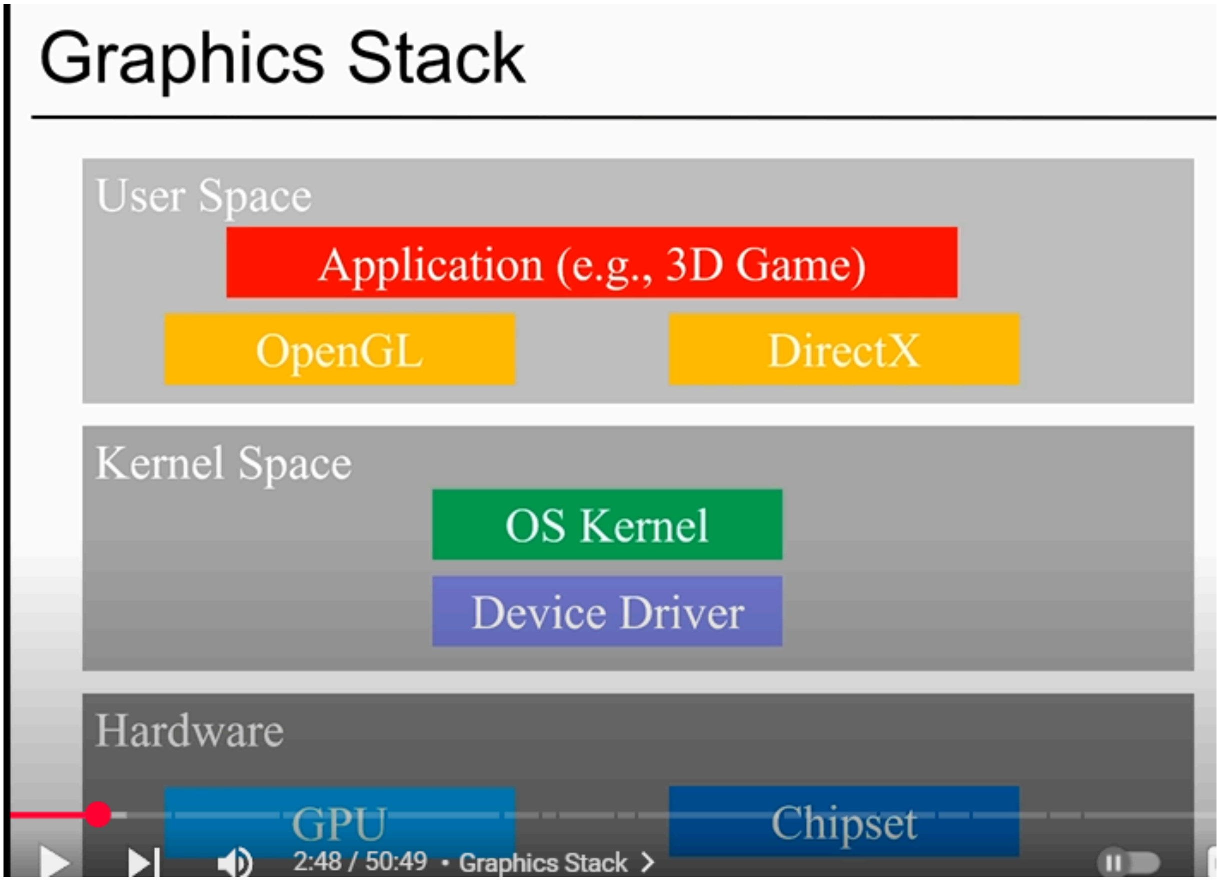
original image

1px median filter

3px median filter

10px median filter

- **L22: Output Devices**
  - Graphics Stack



Graphics Stack

User Space

Application (e.g., 3D Game)

OpenGL

DirectX

Kernel Space

OS Kernel

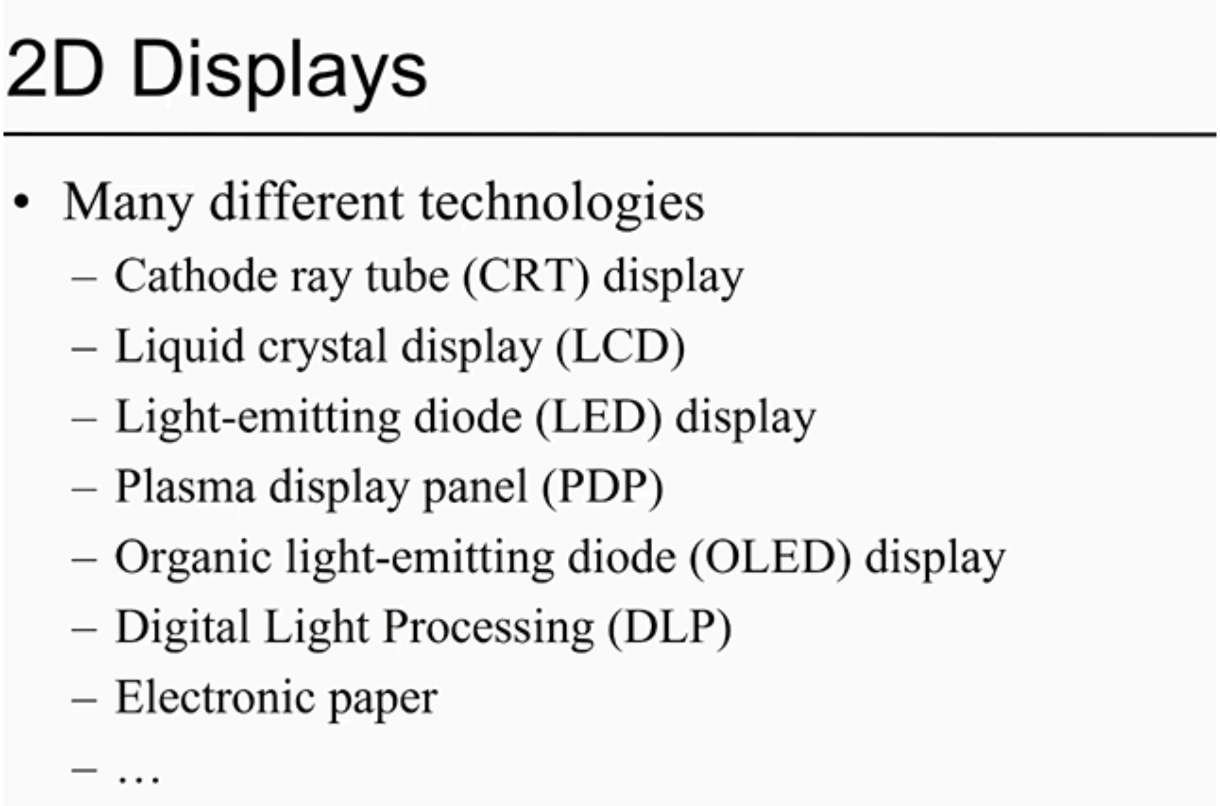
Device Driver

Hardware

GPU

Chipset

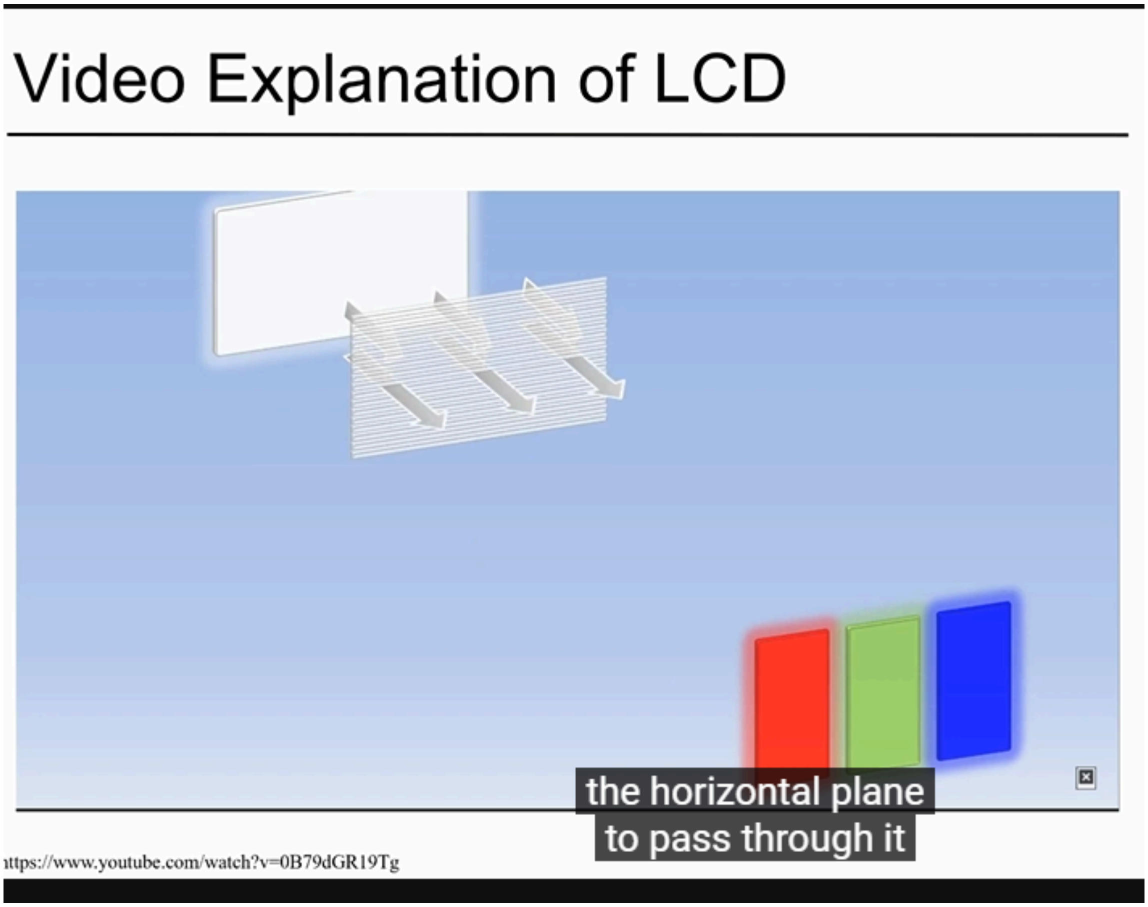
- 2D Displays



2D Displays

- Many different technologies
  - Cathode ray tube (CRT) display
  - Liquid crystal display (LCD)
  - Light-emitting diode (LED) display
  - Plasma display panel (PDP)
  - Organic light-emitting diode (OLED) display
  - Digital Light Processing (DLP)
  - Electronic paper
  - …

  - CRT Display
  - LCD (Liquid Crystal Displays)



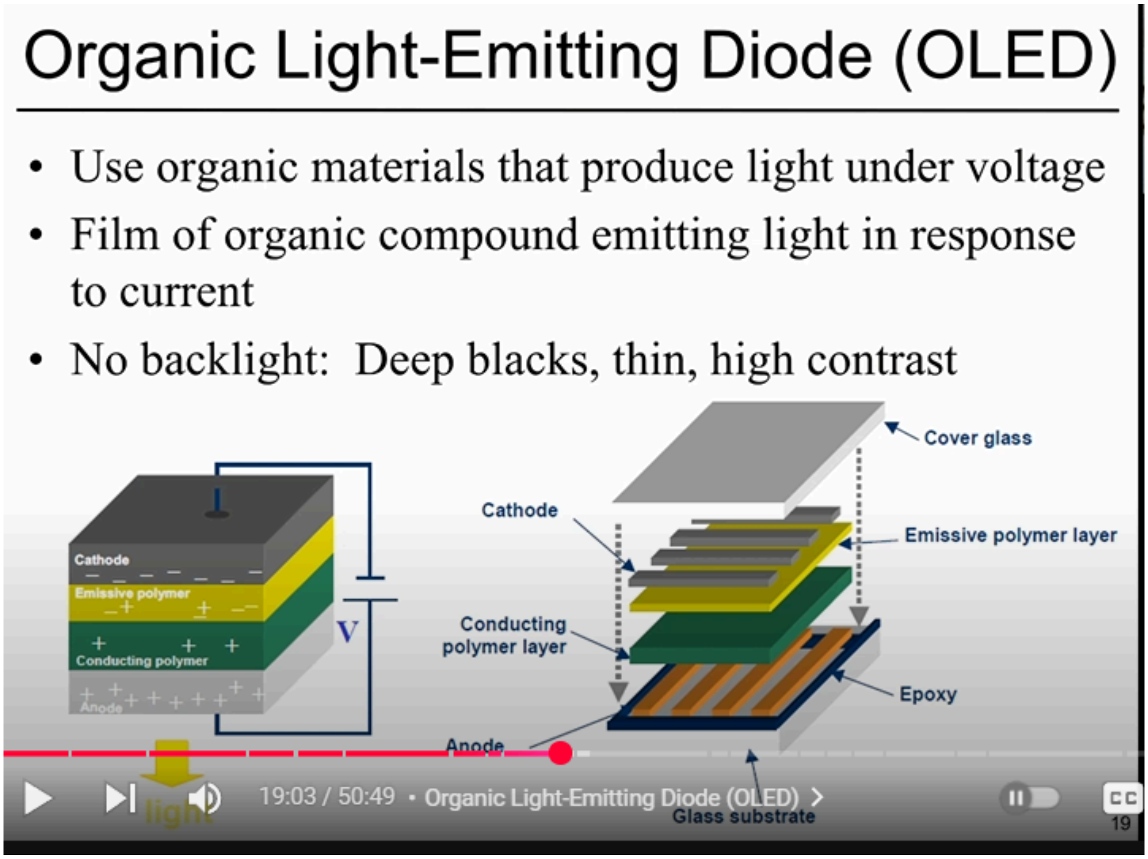
Video Explanation of LCD

the horizontal plane
to pass through it

ttps://www.youtube.com/watch?v=0B79dGR19Tg

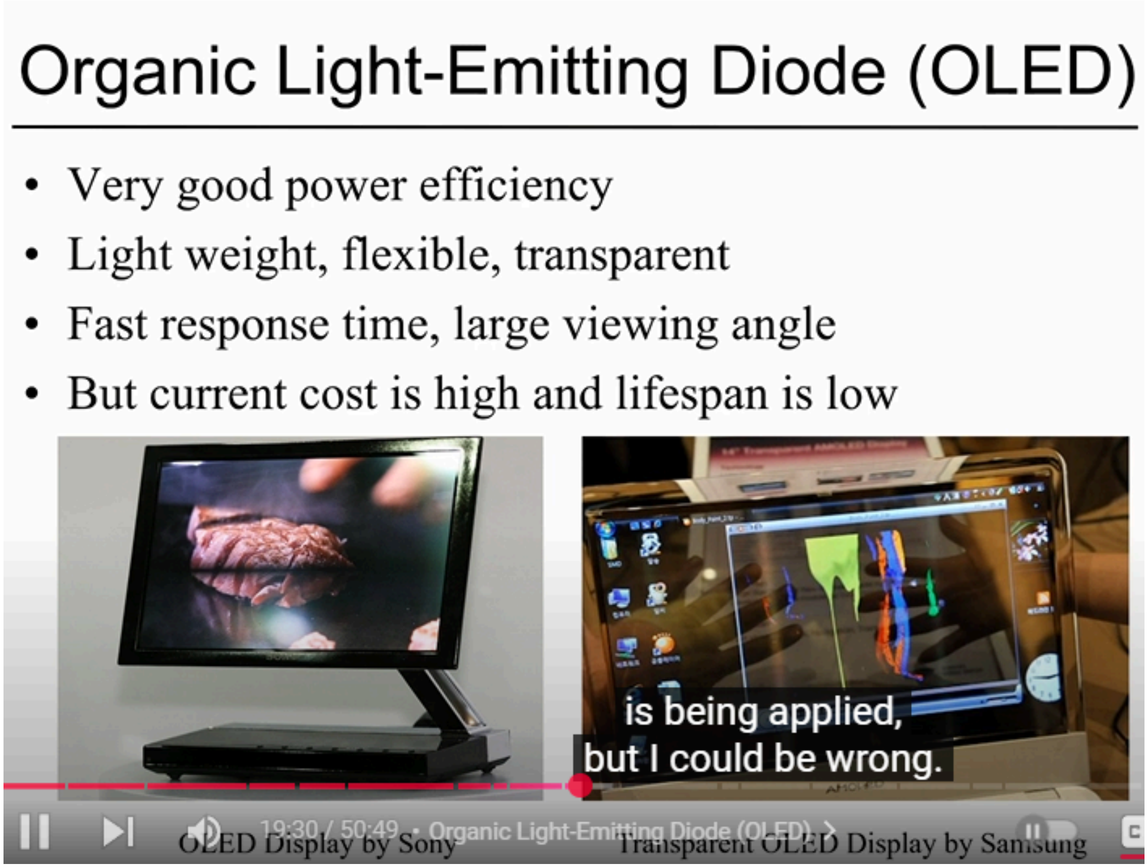
  - LED (Light-Emitting Diode)

- PDP (Plasma Display Panels)
- OLED (Organic Light-Emitting Diode)



Organic Light-Emitting Diode (OLED)

- Use organic materials that produce light under voltage
- Film of organic compound emitting light in response to current
- No backlight: Deep blacks, thin, high contrast



Organic Light-Emitting Diode (OLED)

- Very good power efficiency
- Light weight, flexible, transparent
- Fast response time, large viewing angle
- But current cost is high and lifespan is low

is being applied,
but I could be wrong.

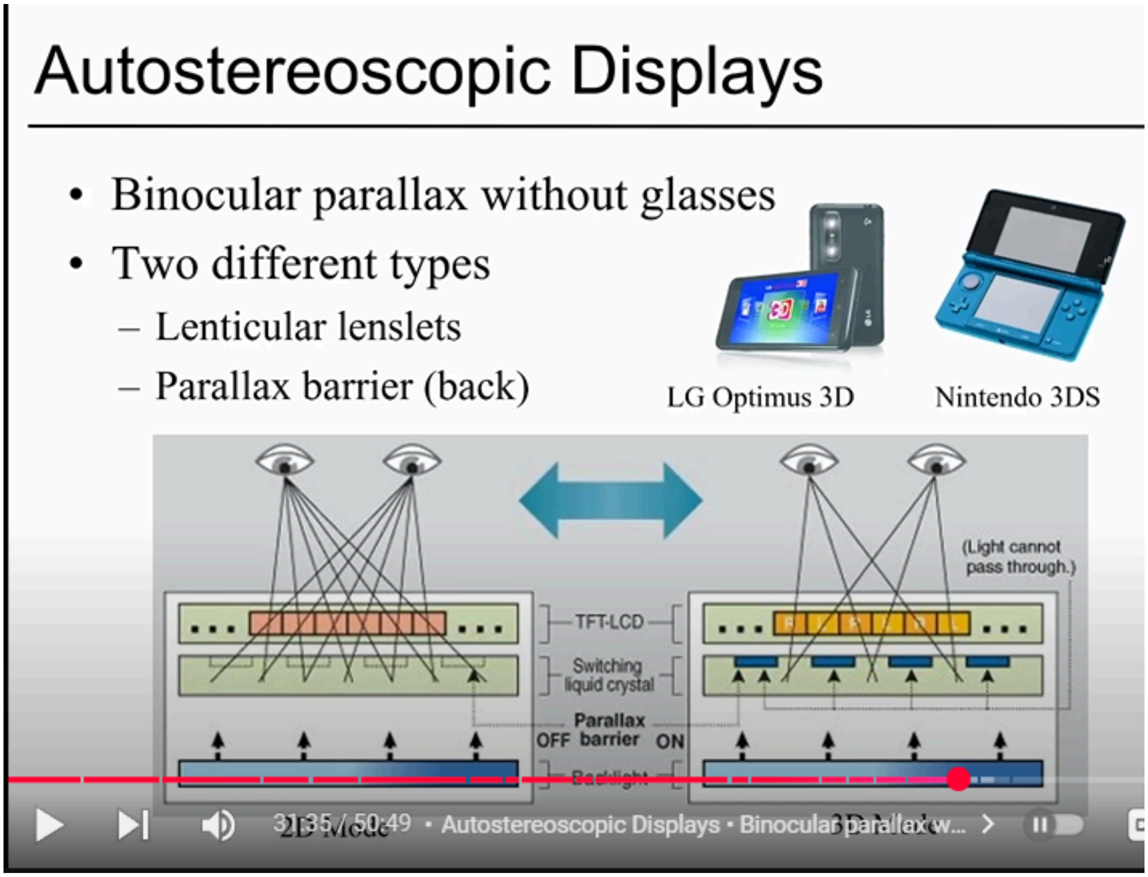
OLED Display by Sony

Transparent OLED Display by Samsung

- DLP (Digital Light Processing)

- 3D Displays

    - Binocular Vision - Stereopsis
    - Depth Perception
    - Autostereoscopic Displays



Autostereoscopic Displays

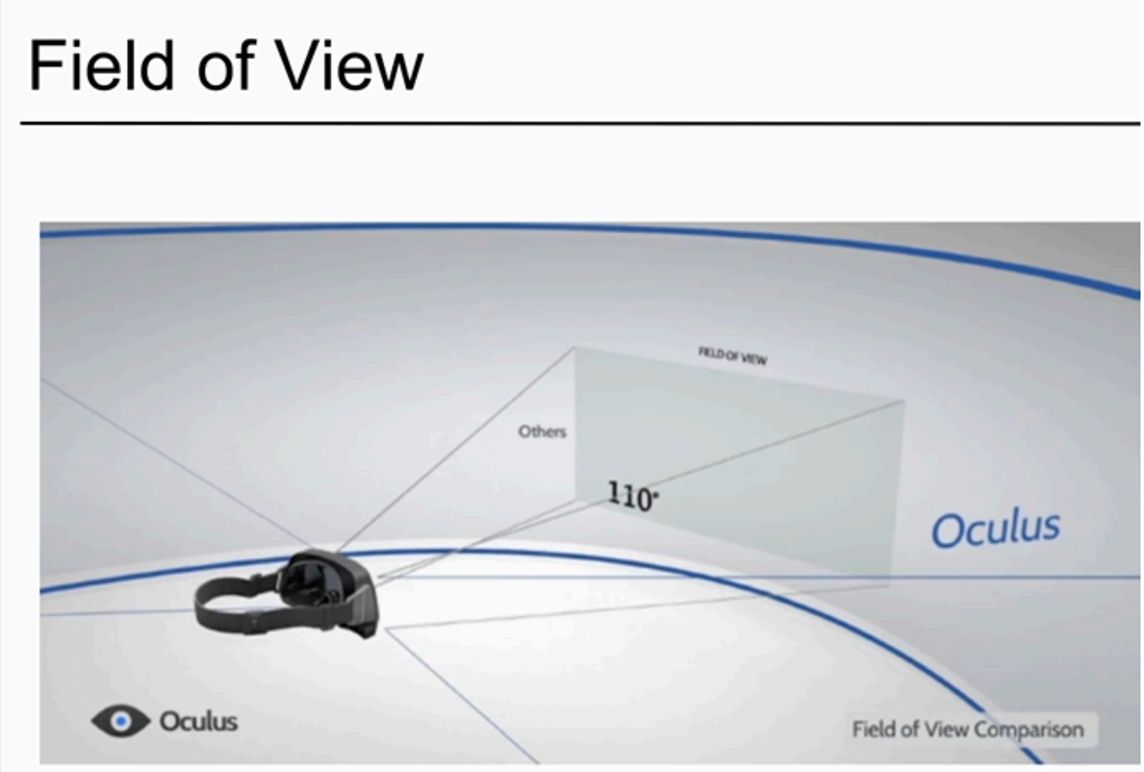
- Binocular parallax without glasses
- Two different types
  - Lenticular lenslets
  - Parallax barrier (back)

LG Optimus 3D

Nintendo 3DS

(Light cannot pass through.)
TFT-LCD
Switching liquid crystal
Parallax barrier
OFF
ON
Backlight
2D Mode
3D Mode
31:35 / 50:49 • Autostereoscopic Displays • Binocular parallax w...

- Virtual Reality & Augmented Reality Displays
    - Field of View



Field of View