CSC418: Computer Graphics

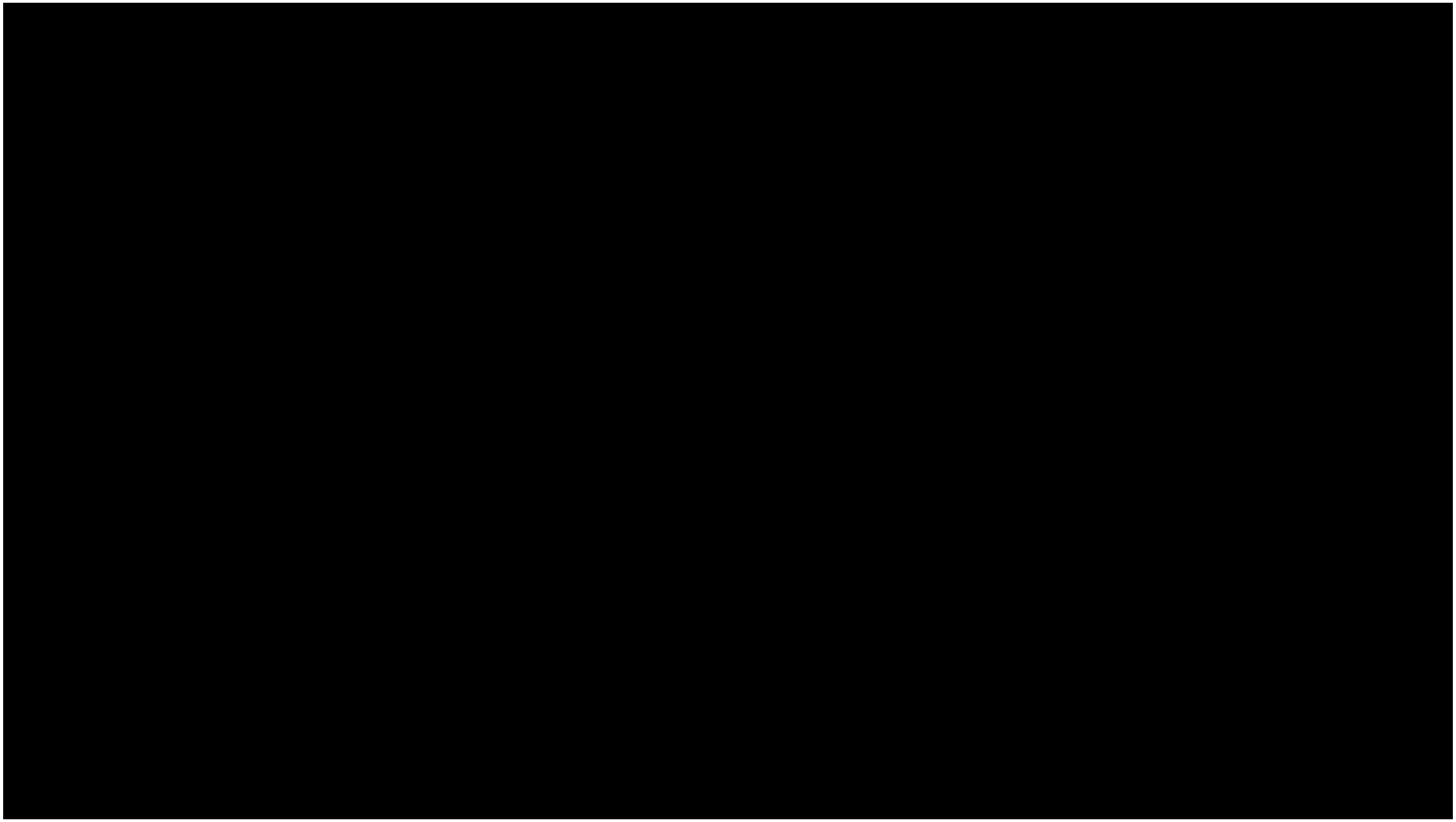
DAVID LEVIN

Today's Topics

1. Texture mapping
2. Ray Tracing

Some slides and figures courtesy of Wolfgang Hürst, Patricio Simari
Some figures courtesy of Peter Shirley,
"Fundamentals of Computer Graphics", 3rd Ed.

Showtime



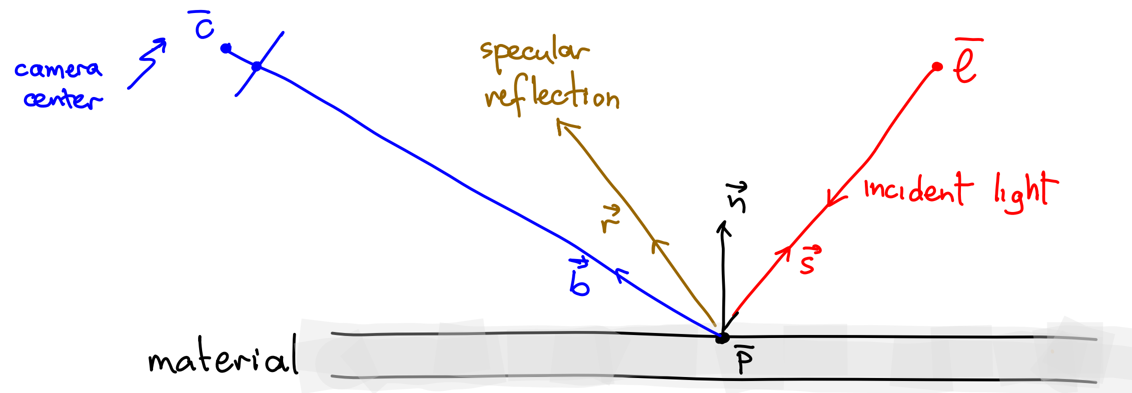
But First … Logistical Things

- You should all have your Assignment 1 and Midterm Grades
- Assignment 2 is due this Friday
  - If you are still having troubles email the TAs
    - csc418tas@cs.toronto.edu
- Karan is still away so email me if you have any issues
  - diwlevin@cs.toronto.edu (usually requires two emails)

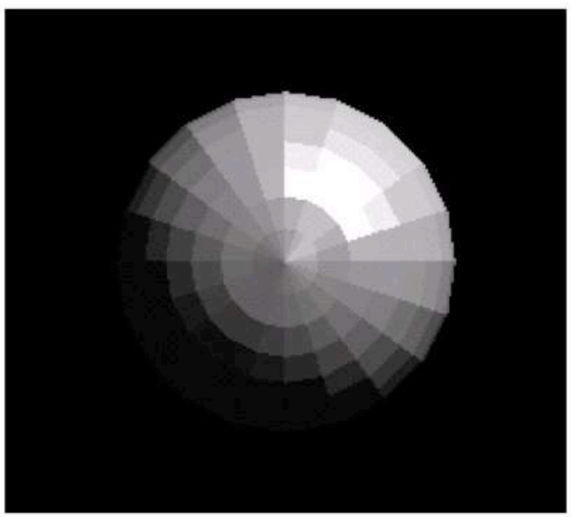
Phong Shading: Comparisons

<u>Phong shading:</u>

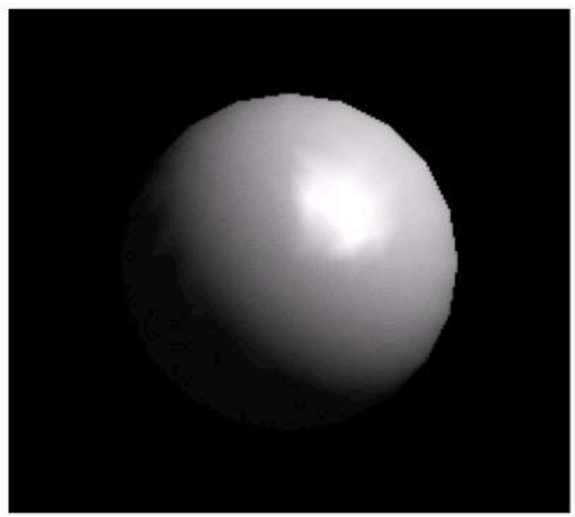
1. Interpolate $\vec{b}_i, \vec{n}_i, \vec{s}_i$ to get $\vec{b}, \vec{n}, \vec{s}$ at $\bar{p}$
2. Compute $L(\vec{b}, \vec{n}, \vec{s})$



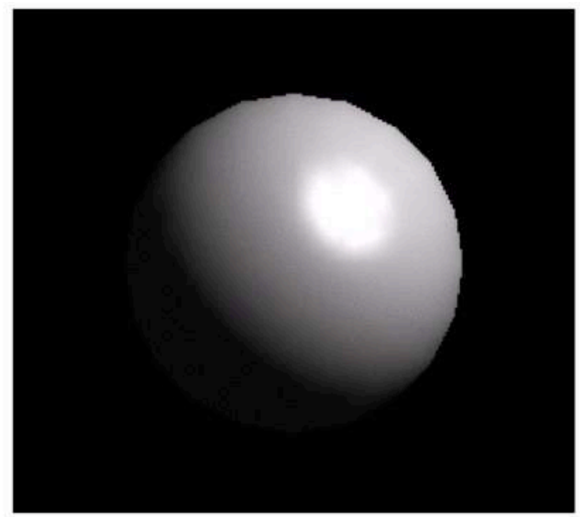
Hsien-Hsin Sean Lee, GaTech



Flat shading



Gouraud shading

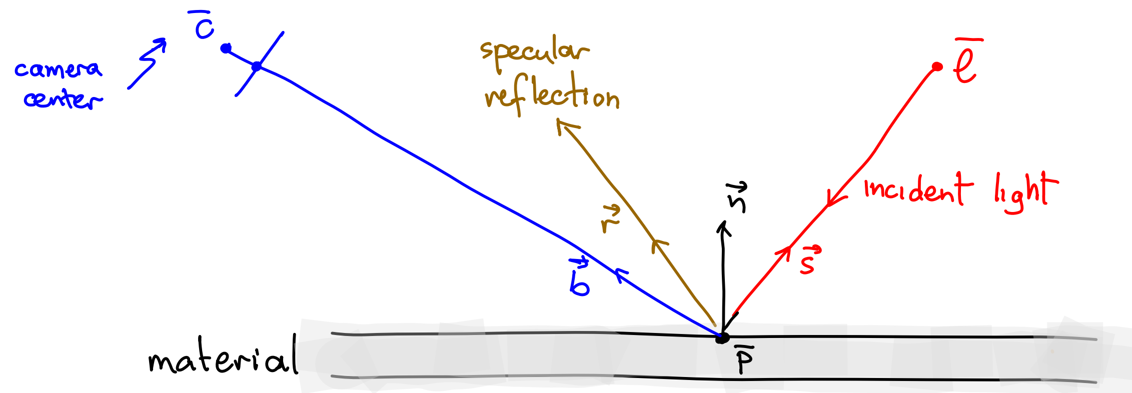


Phong shading

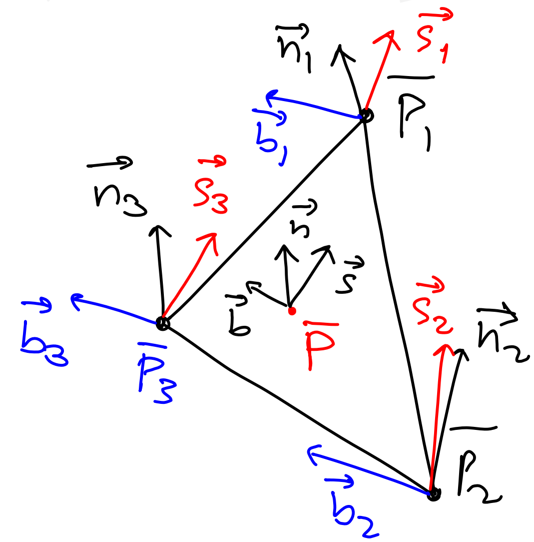
Phong Shading: Comparisons

Phong shading:

1. Interpolate $\vec{b}_i, \vec{n}_i, \vec{s}_i$ to get $\vec{b}, \vec{n}, \vec{s}$ at $\bar{p}$
2. Compute $L(\vec{b}, \vec{n}, \vec{s})$



Comparison to Gouraud shading



\+ Smooth intensity variations as in Gouraud shading

\+ Handles specular highlights correctly even for large triangles (Why?)

\- Computationally less efficient (but okay in today's hardware!) (Must interpolate 3 vectors & evaluate Phong reflection model at each triangle pixel)

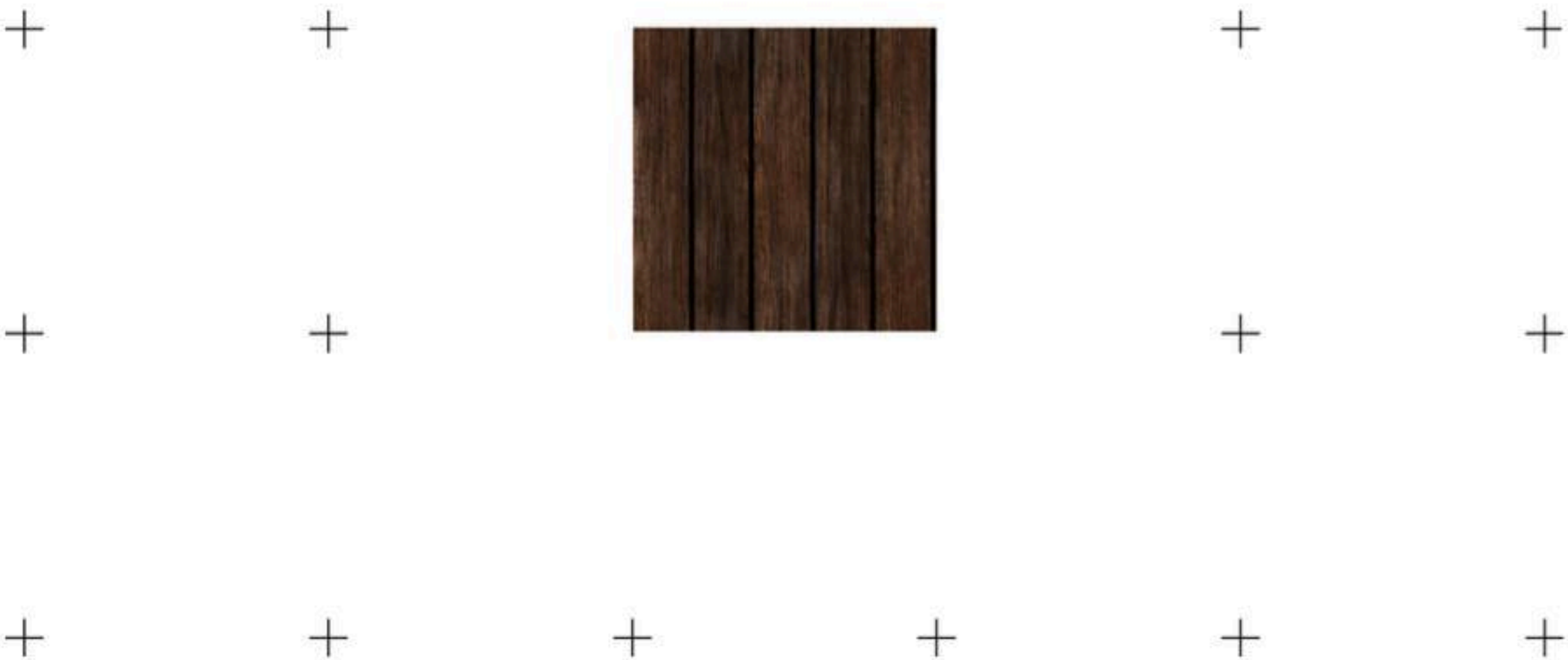
Topic 1:

Texture Mapping

- Motivation
- Sources of texture
- Texture coordinates
- {Bump, MIP, displacement, environmental} mapping

Motivation

- Adding lots of detail to our models to realistically depict skin, grass, bark, stone, etc., would increase rendering times dramatically, even for hardware-supported projective methods.



Motivation

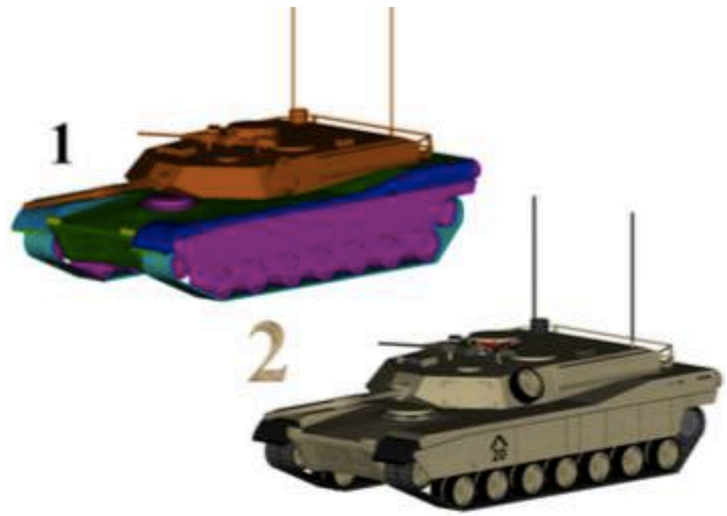
- Adding lots of detail to our models to realistically depict skin, grass, bark, stone, etc., would increase rendering times dramatically, even for hardware-supported projective methods.

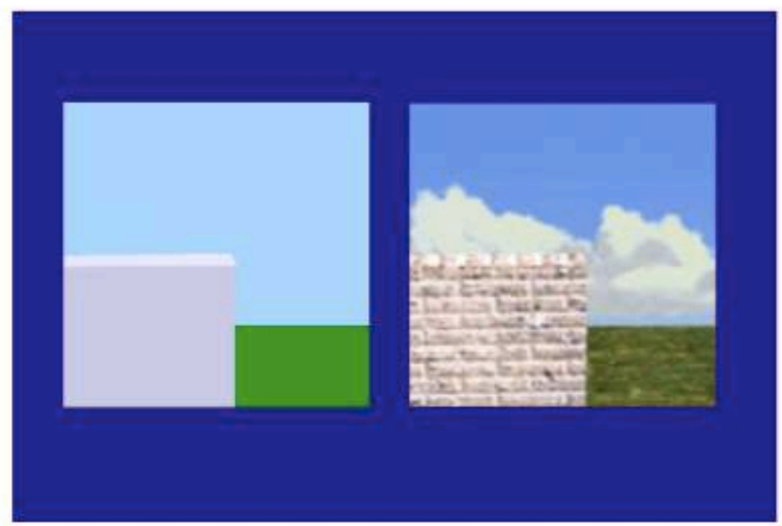
Motivation

Basic idea of texture mapping:

Instead of calculating color, shade, light, etc. for each pixel we just paste images to our objects in order to create the illusion of realism

Different approaches exist (e.g. tiling; cf. previous slide)





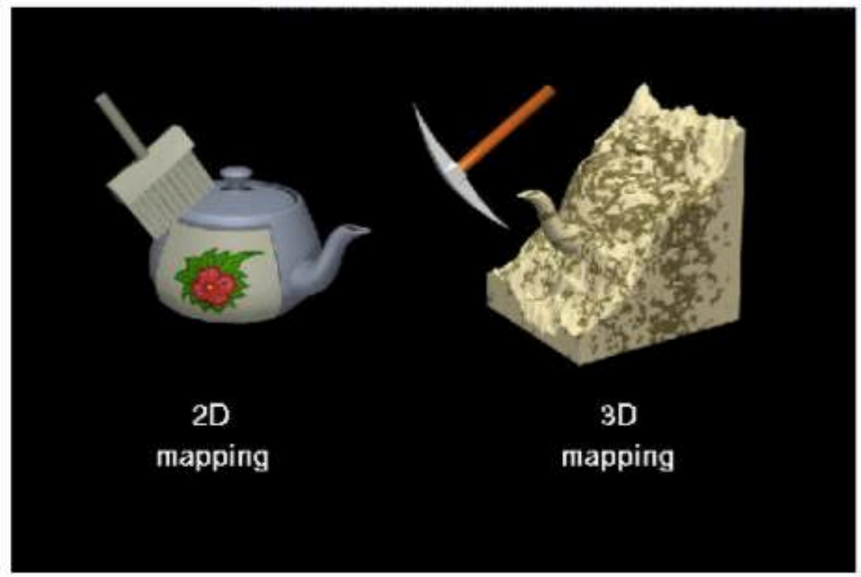
Motivation

In general, we distinguish between 2D and 3D texture mapping:

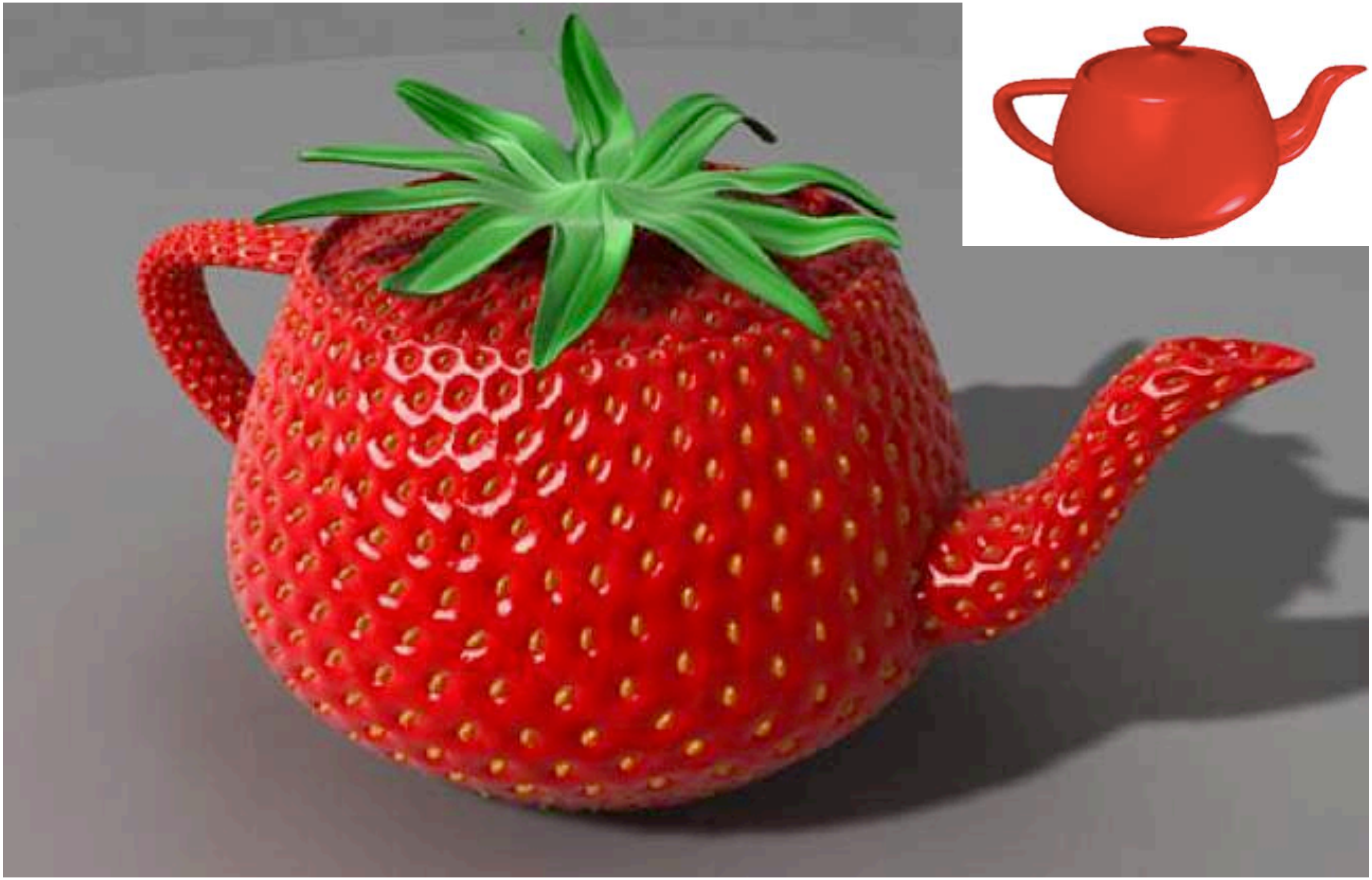
2D mapping (aka *image textures*): paste an image onto the object

3D mapping (aka *solid* or *volume textures*): create a 3D texture and "carve" the object

3D Object



2D texture $\longleftrightarrow$ 3D texture

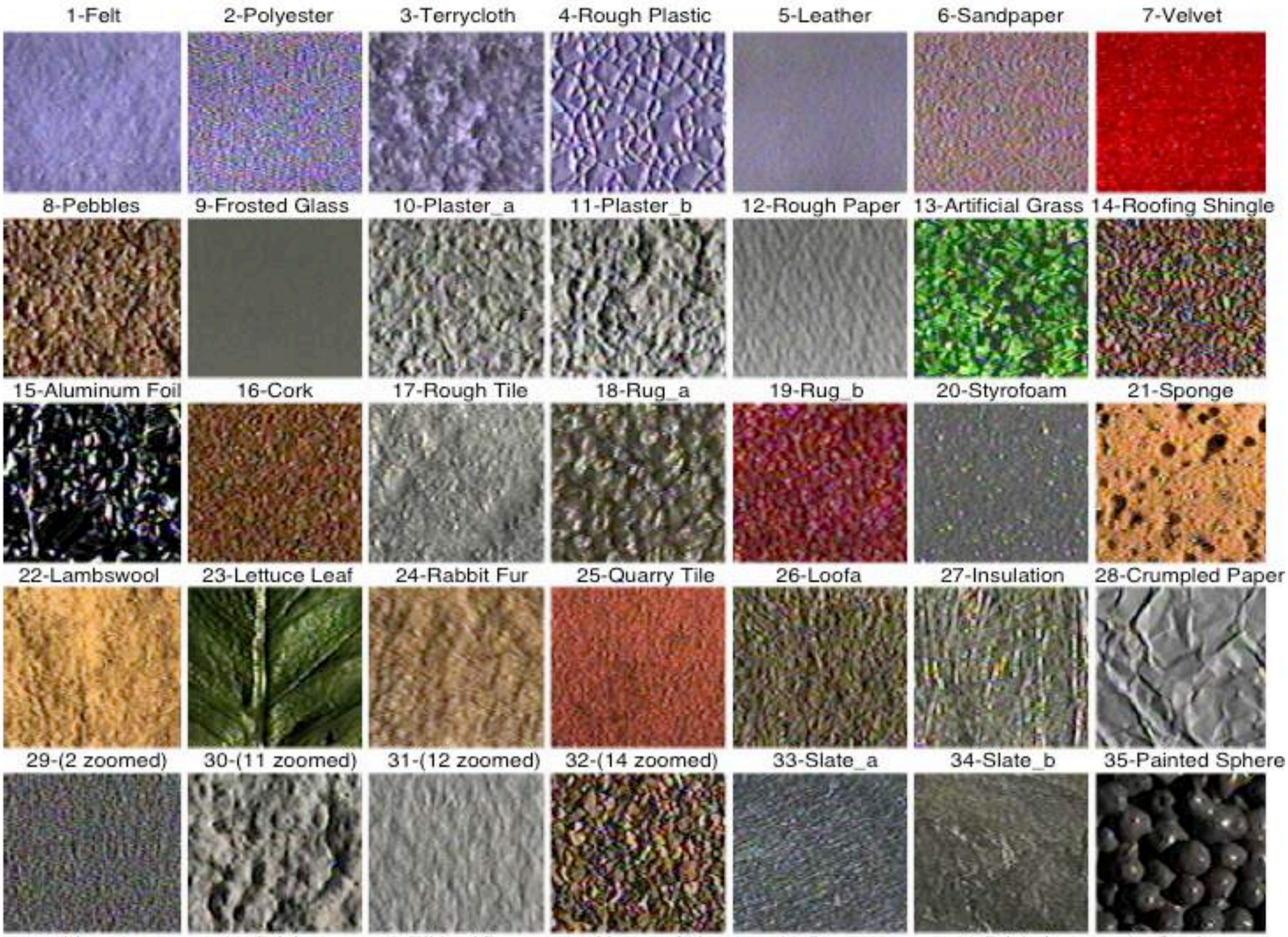


Topic 1:

Texture Mapping

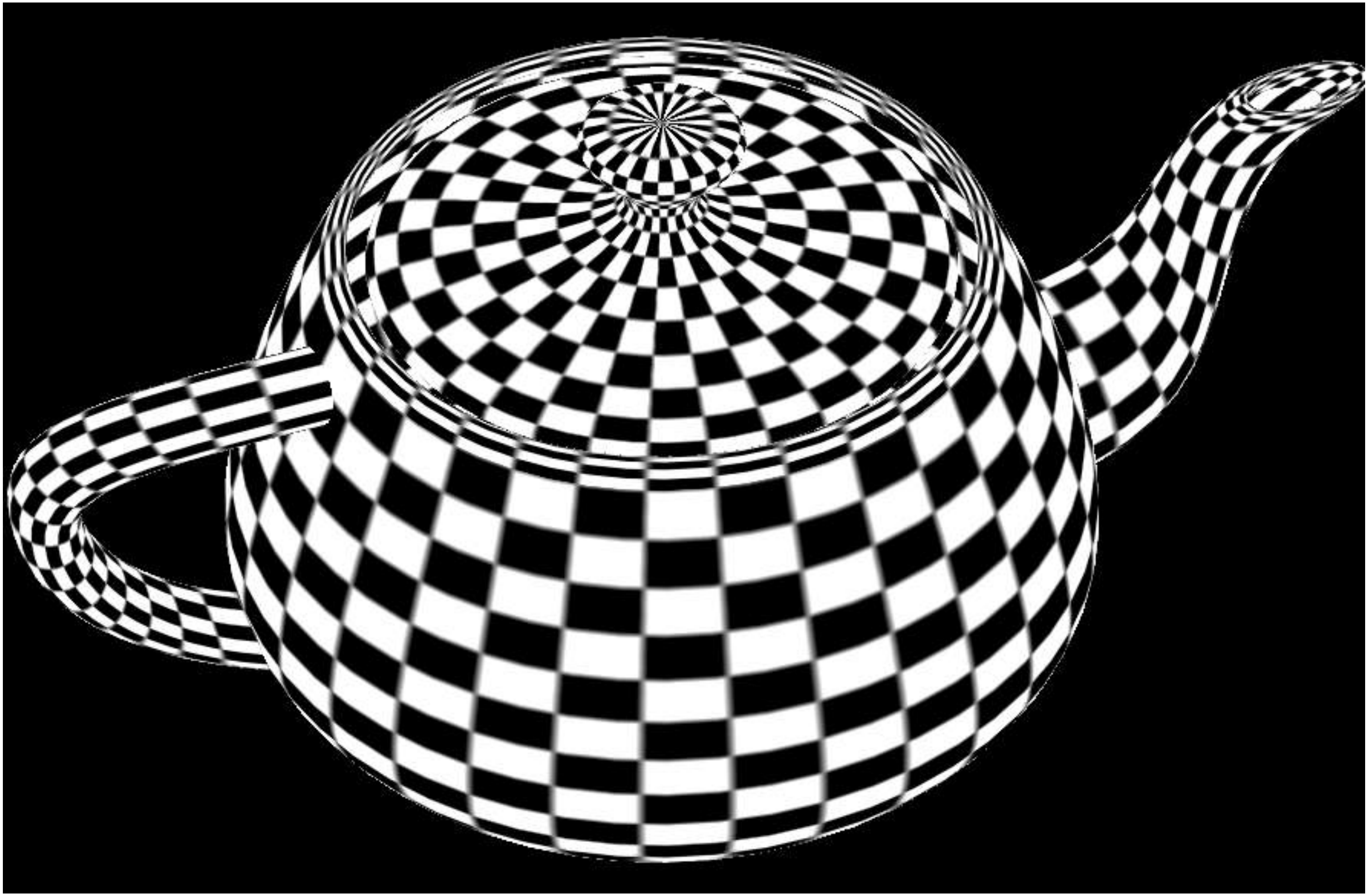
- Motivation
- Sources of texture
- Texture coordinates
- {Bump, MIP, displacement, environmental} mapping

Texture sources: Photographs

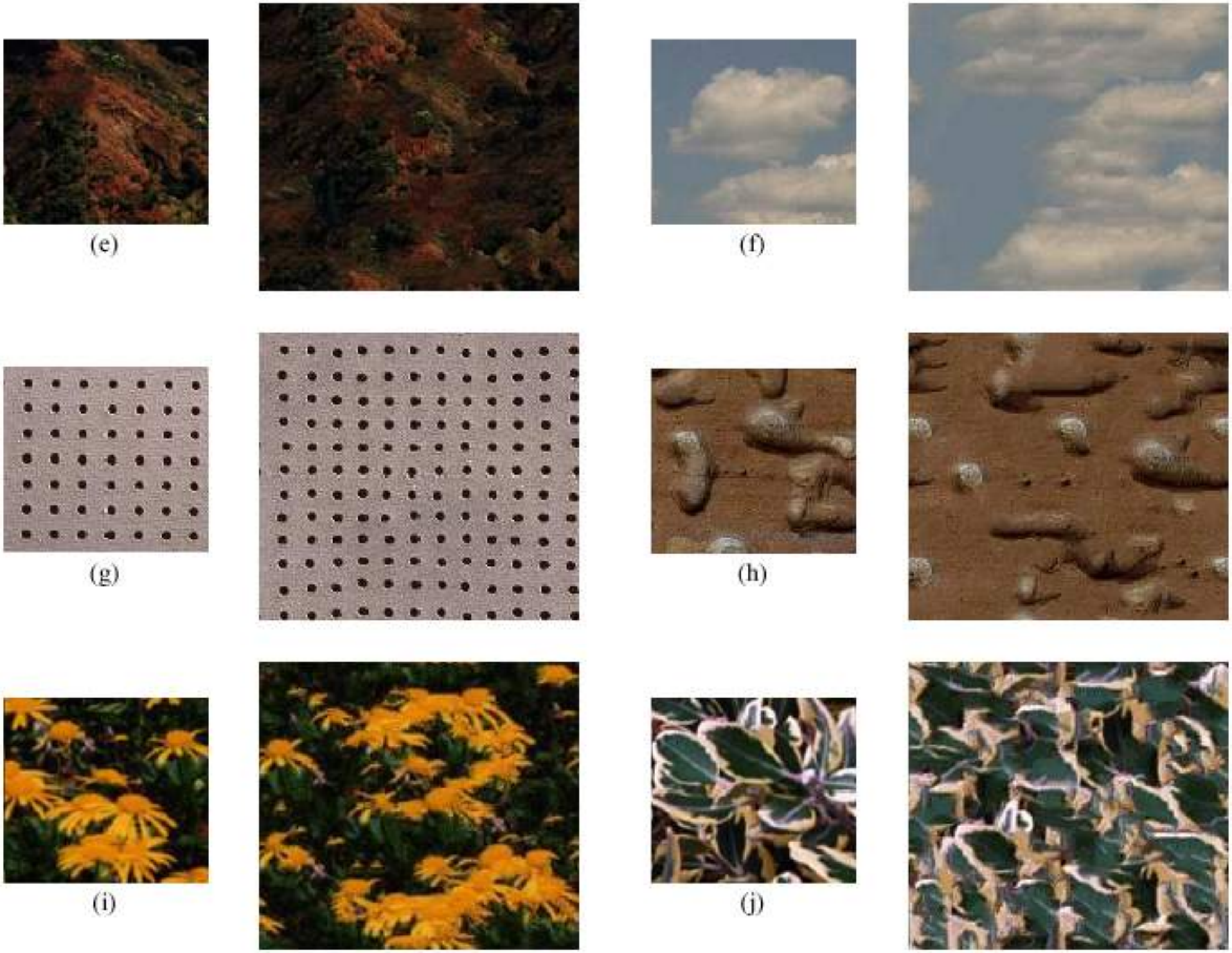


Texture sources: Solid textures

Texture sources: Procedural



Texture sources: Synthesized



Kwatra et al, SIGGRAPH '05

Original

Synthesized

Original

Synthesized

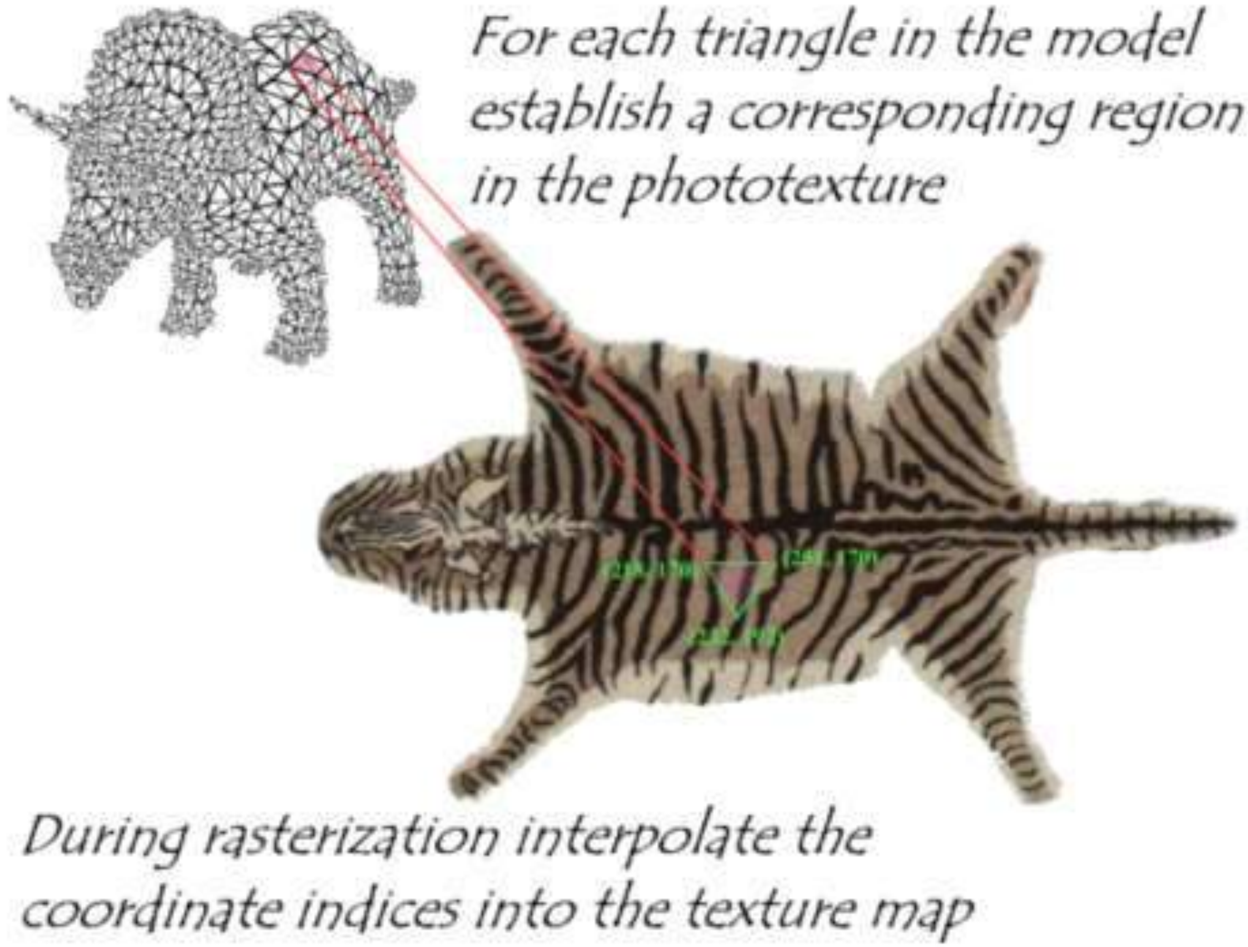
Topic 1:

Texture Mapping

- Motivation
- Sources of texture
- Texture coordinates
- {Bump, MIP, displacement, environmental} mapping

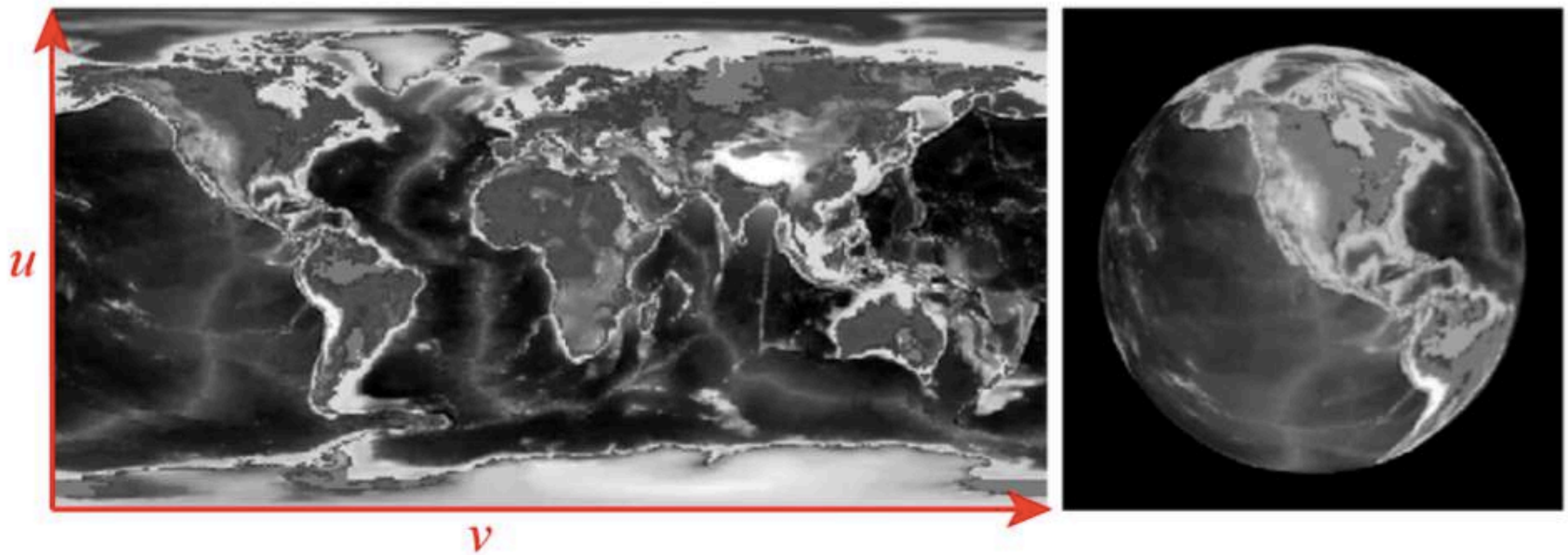
Texture coordinates

How does one establish correspondence? (UV mapping)



Texture coordinates

Example: use world map and sphere to create a globe



Per conventions we usually assume $u, v \in [0, 1]$.

Texture coordinates

$$\begin{aligned} x &= x_c + r\cos\phi\sin\theta \\ y &= y_c + r\sin\phi\sin\theta \\ z &= z_c + r\cos\theta \end{aligned}$$

Given a point (x, y, z) on the surface of the sphere, we can find θ and ϕ by

$$\begin{aligned} \theta &= \arccos\tfrac{z-z_c}{r} \qquad \text{(cf. longitude)} \\ \phi &= \arctan\tfrac{y-y_c}{x-x_c} \qquad \text{(cf. latitude)} \end{aligned}$$

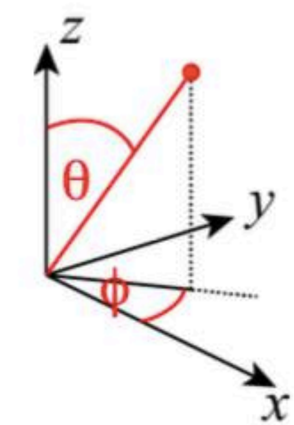
(Note: $\arccos$ is the inverse of $\cos$, $\arctan$ is the inverse of $\tan = \frac{\sin}{\cos}$)

Texture coordinates

For a point (x, y, z) we have

$$\theta = \arccos \frac{z - z_c}{r}$$
$$\phi = \arctan \frac{y - y_c}{x - x_c}$$

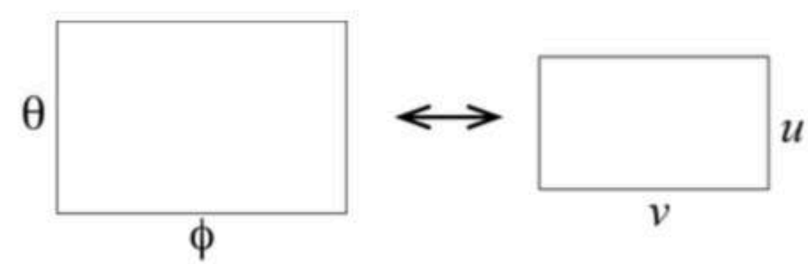
$(\theta, \phi) \in [0, \pi] \times [-\pi, \pi]$, and u, v must range from $[0, 1]$.



Hence, we get:

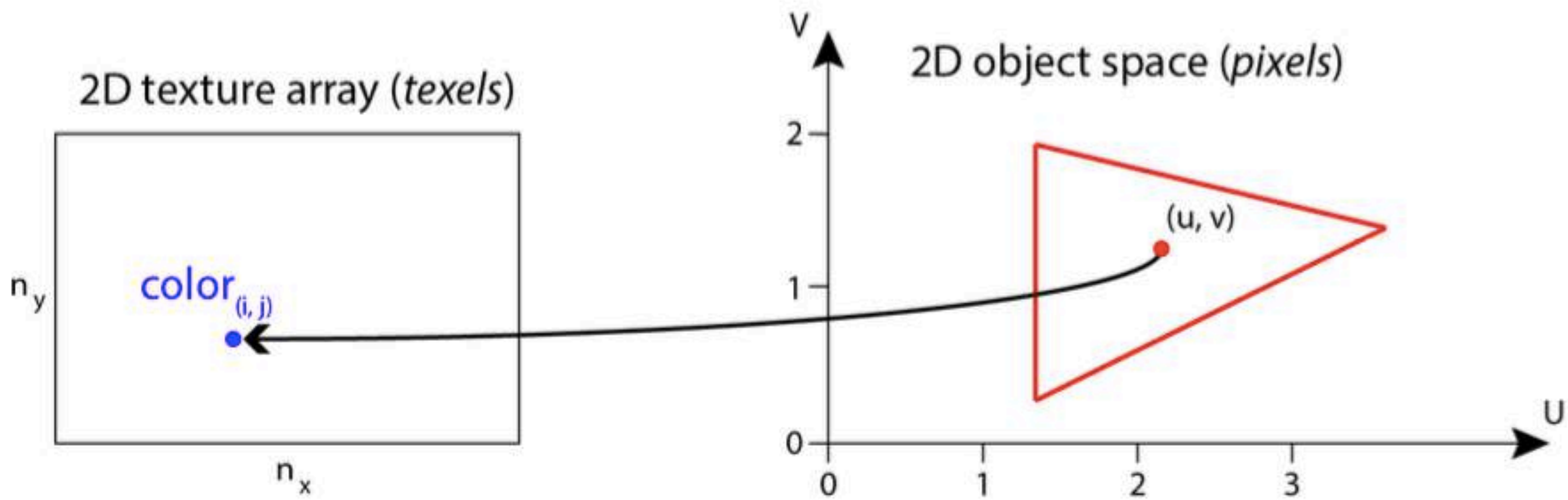
$$u = \frac{\phi \mod 2\pi}{2\pi}$$
$$v = \frac{\pi - \theta}{\pi}$$

(Note that this is a simple scaling transformation in 2D)



Texture coordinates

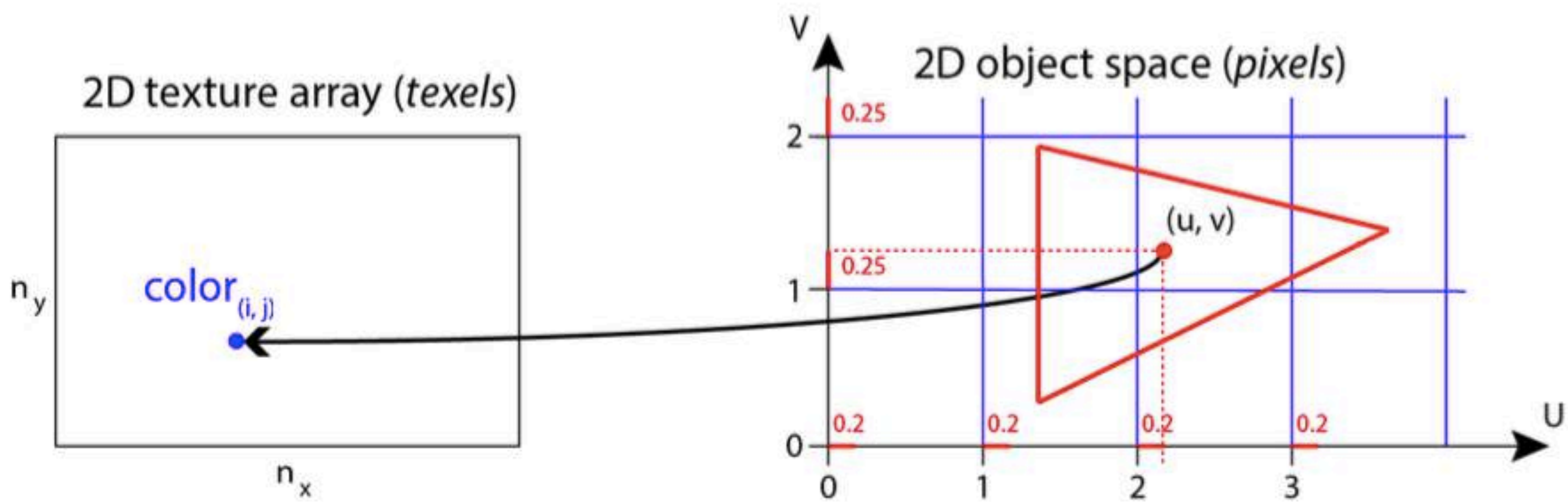
Example: "Tiling" of 2D textures into a UV-object space



We'll call the two dimensions to be mapped u and v,
and assume an $n_x \times n_y$ image as texture.

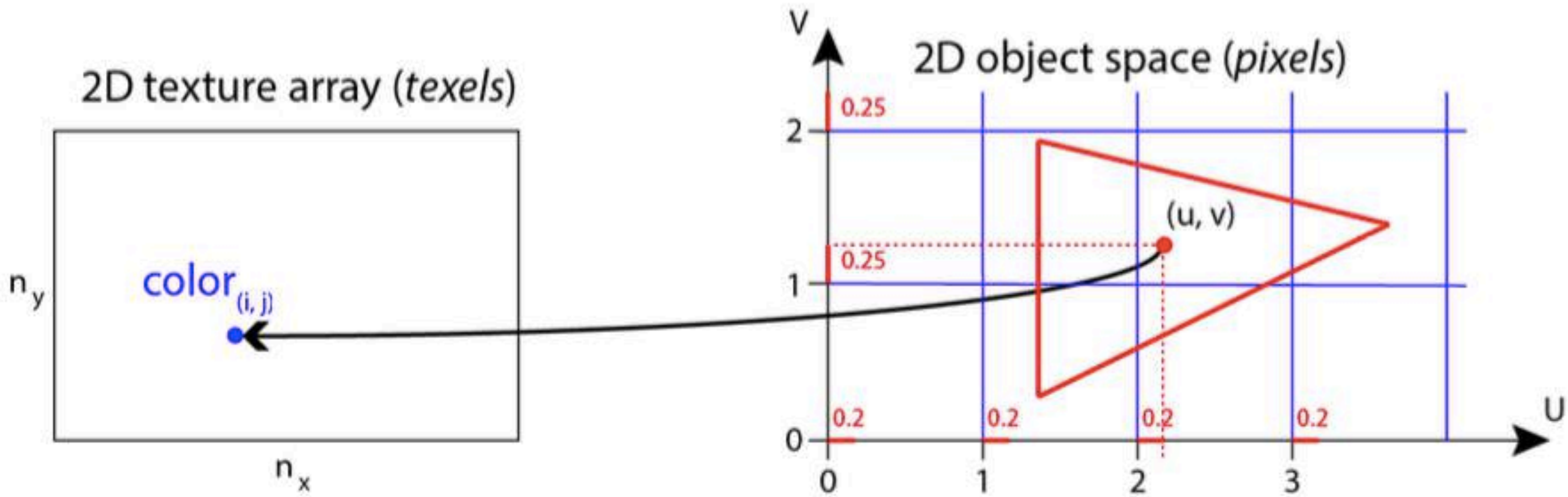
Then every (u, v) needs to be mapped to a color in the image,
i.e. we need a mapping from pixels to texels.

Texture coordinates



A standard way is to first remove the integer portion of u and v, so that (u, v) lies in the unit square.

Texture coordinates

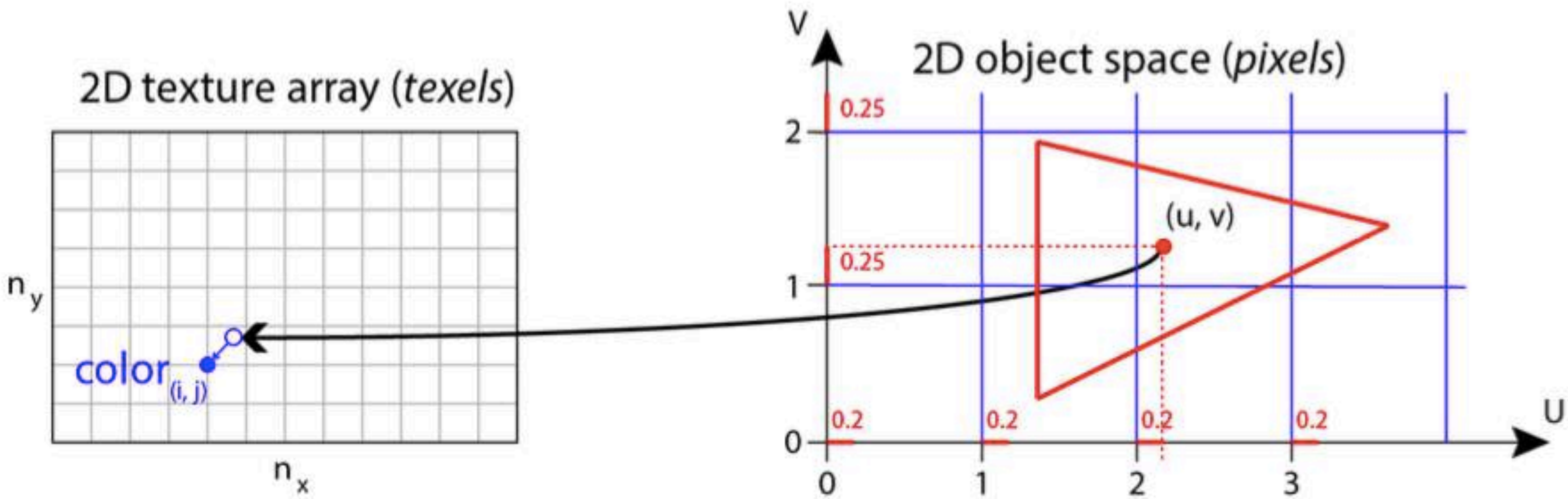


This results in a simple mapping from $0 \leq u, v \leq 1$ to the size of the texture array, i.e. $n_x \times n_y$.

$$i = un_x \text{ and } j = vn_y$$

Yet, for the array lookup, we need integer values.

Texture coordinates



The texel (i, j) in the $n_x \times n_y$ image for (u, v) can be determined using the floor function $\lfloor x \rfloor$ which returns the highest integer value $\leq x$.

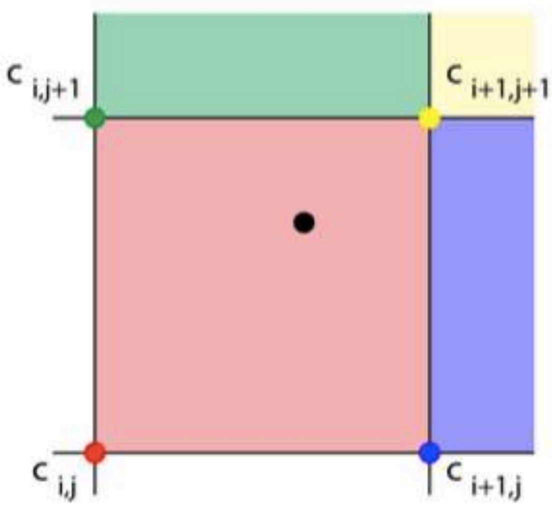
$$i = \lfloor u n_x \rfloor \text{ and } j = \lfloor v n_y \rfloor$$

Texture coordinates

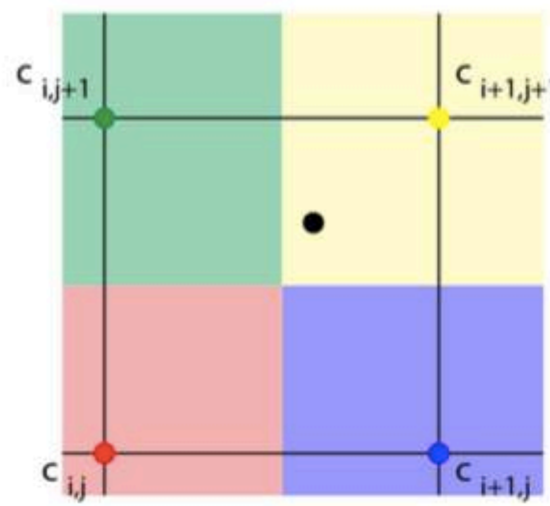
$$c(u,v) = c_{i,j} \text{ with } i = \lfloor un_x \rfloor \text{ and } j = \lfloor vn_y \rfloor$$

This is a version of nearest-neighbor interpolation, where we take the color of the nearest neighbor.

Floor function



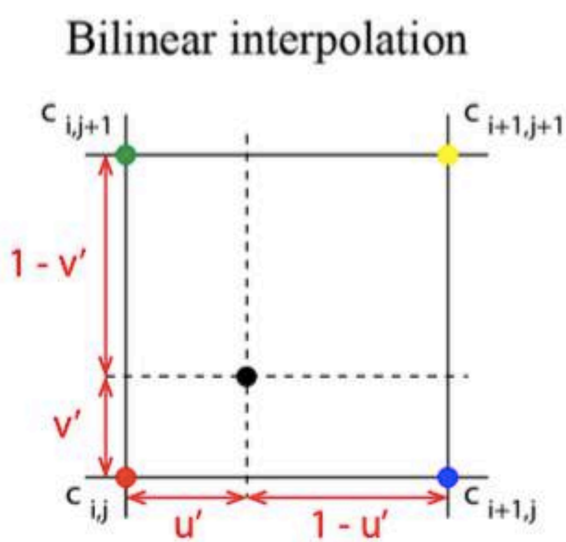
Nearest neighbor mapping



Texture coordinates

For smoother effects we may use bilinear interpolation:

$$c(u,v) =$$
$$(1-u')(1-v')c_{ij}+u'(1-v')c_{(i+1)j}+(1-u')v'c_{i(j+1)}+u'v'c_{(i+1)(j+1)}$$



with
$u' = un_x - \lfloor un_x \rfloor$ and
$v' = vn_y - \lfloor vn_y \rfloor$

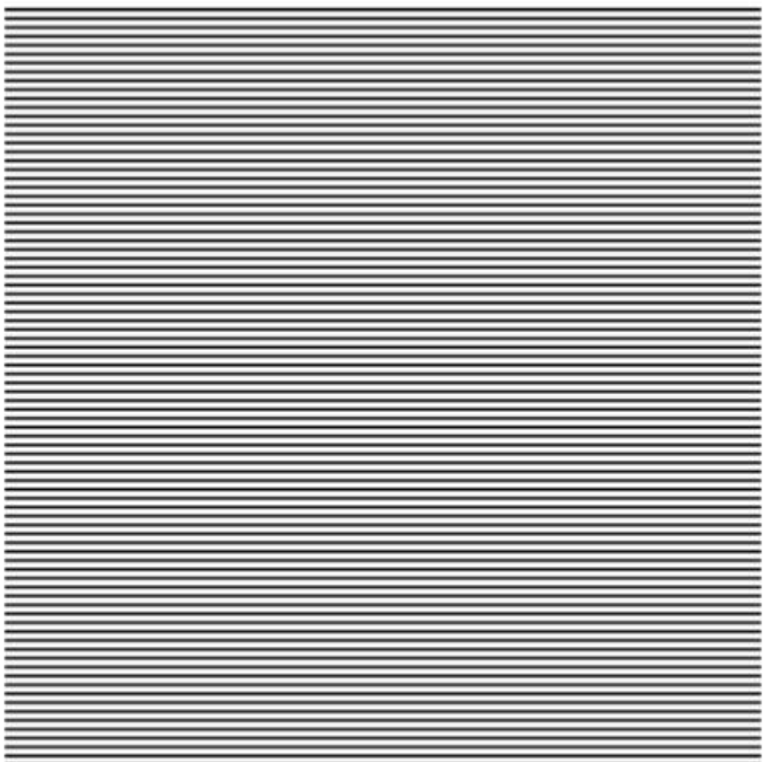
Notice that all weights are between 0 and 1 and add up to 1:
$(1-u')(1-v') + u'(1-v')+$
$(1-u')v' + u'v' = 1$

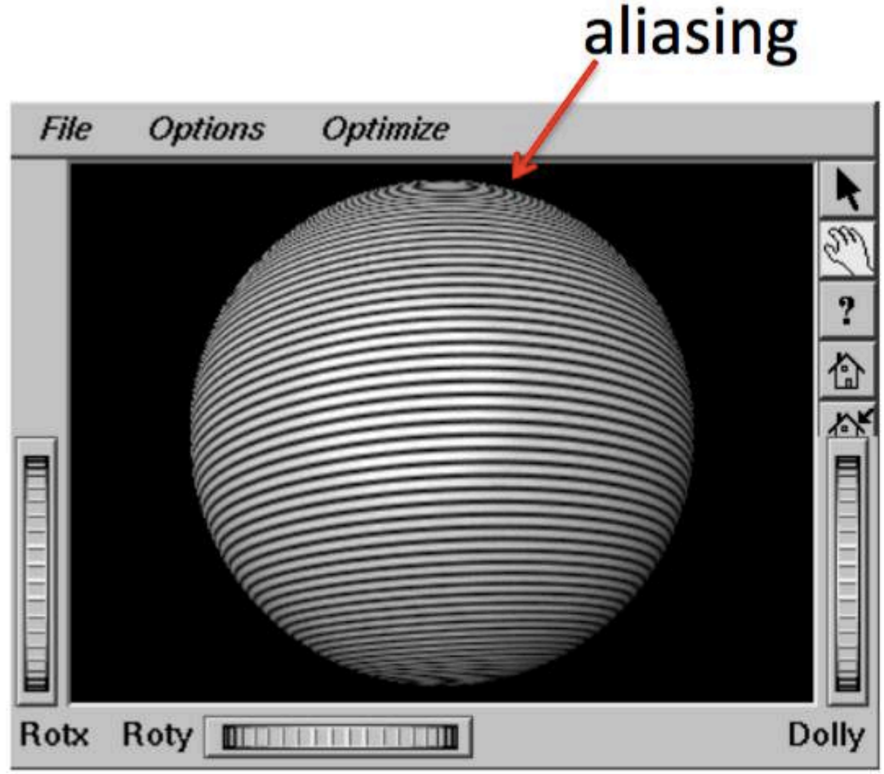
Topic 1:

Texture Mapping

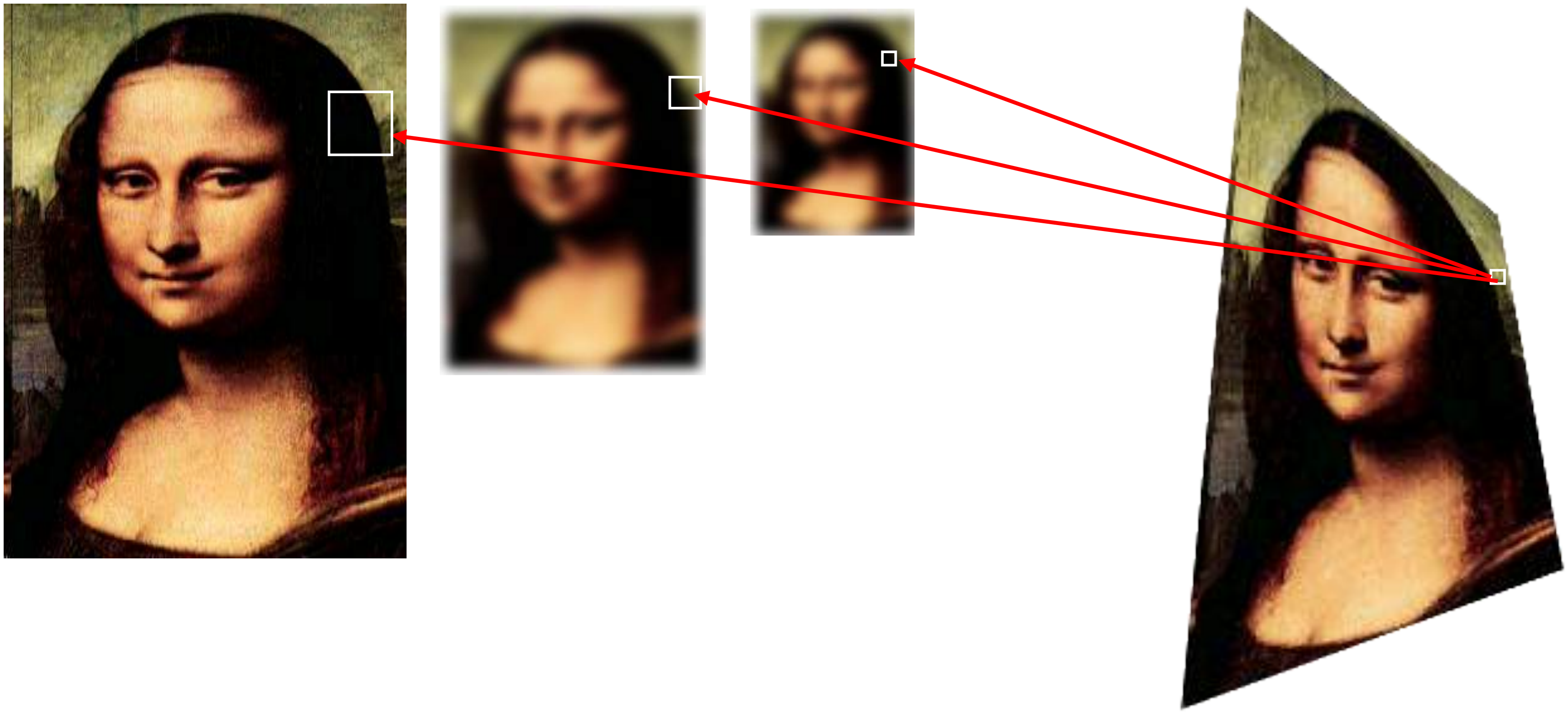
- Motivation
- Sources of texture
- Texture coordinates
- {Bump, MIP, displacement, environmental} mapping

Mipmapping



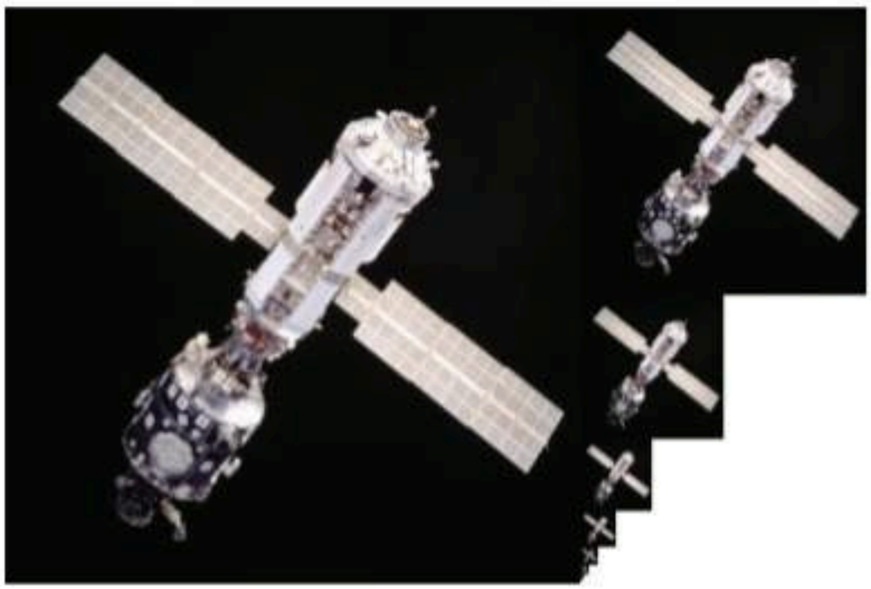


MIP-Mapping: Basic Idea



Given a polygon, use the texture image, where the projected polygon best matches the size of the polygon on screen.

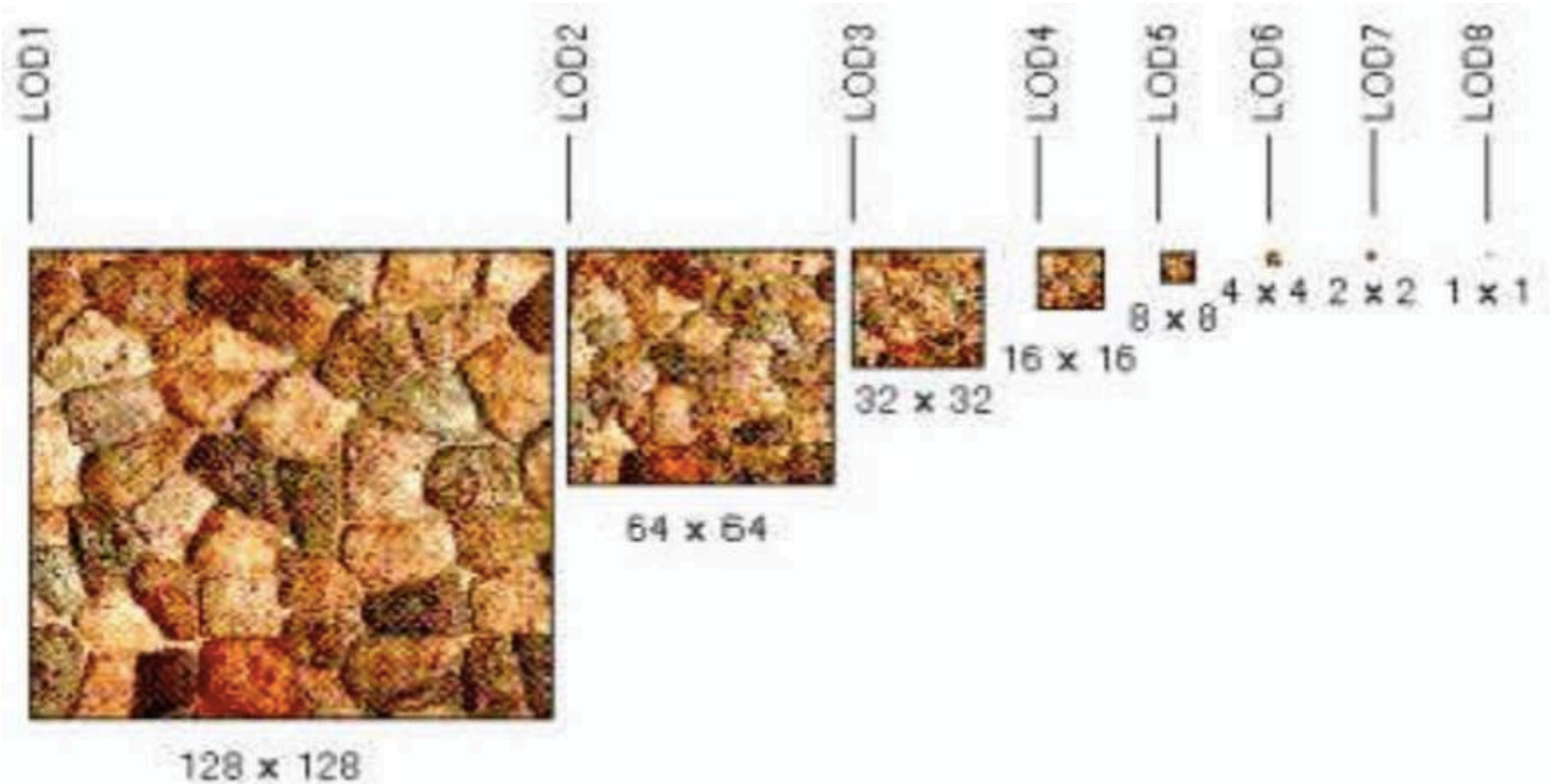
Mipmapping



Solutions: MIP maps

- Pre-calculated, optimized collections of images based on the original texture
- Dynamically chosen based on depth of object (relative to viewer)
- Supported by todays hardware and APIs

Mipmapping



LOD1
LOD2
LOD3
LOD4
LOD5
LOD6
LOD7
LOD8
128 x 128
64 x 64
32 x 32
16 x 16
8 x 8
4 x 4
2 x 2
1 x 1

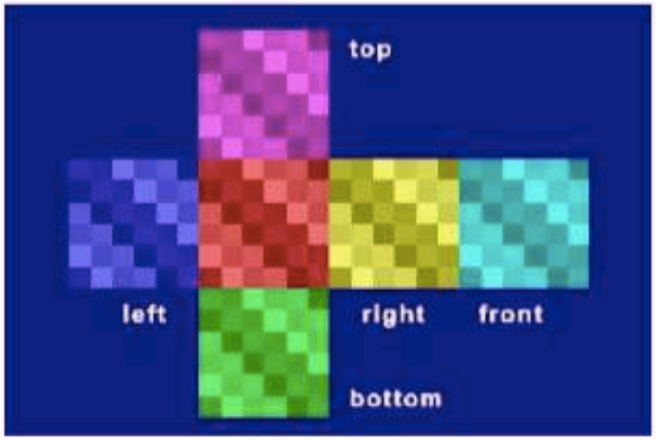
Environment mapping

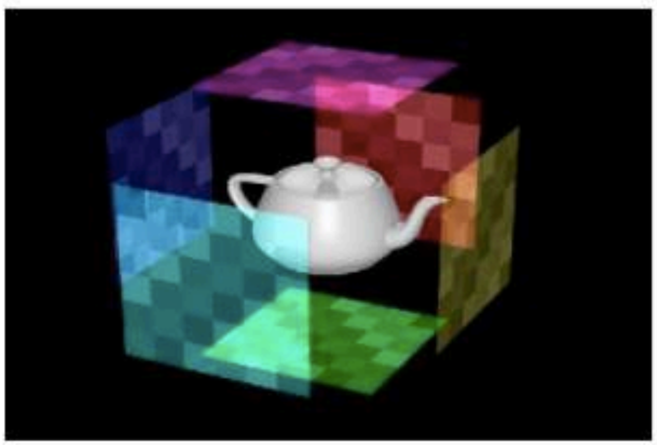
… why not use this to make objects appear to reflect their surroundings specularly?

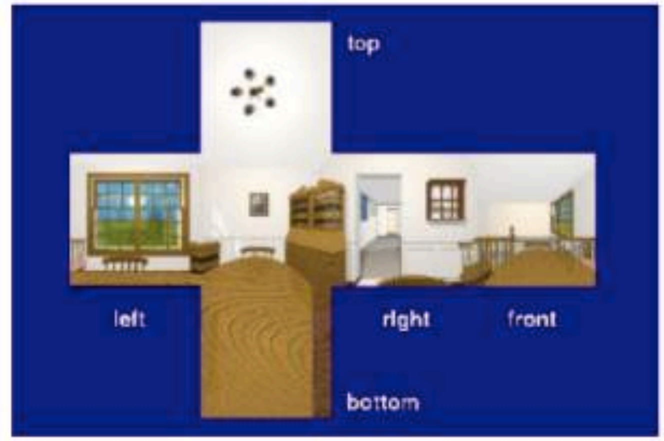
Idea: place a cube around the object, and project the environment of the object onto the planes of the cube in a preprocessing stage; this is our texture map.

During rendering, we compute a reflection vector, and use that to look-up texture values from the cubic texture map.

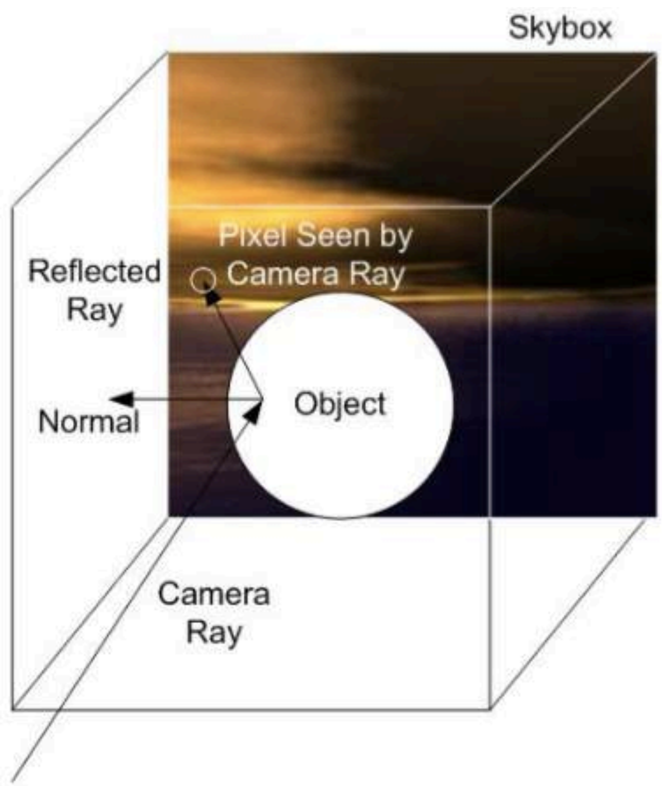
Environment mapping







Environment mapping



Remember Phong shading: “perfect” reflection if

angle between eye vector $\vec{e}$ and $\vec{n}$ = angle between $\vec{n}$ and reflection vector $\vec{r}$

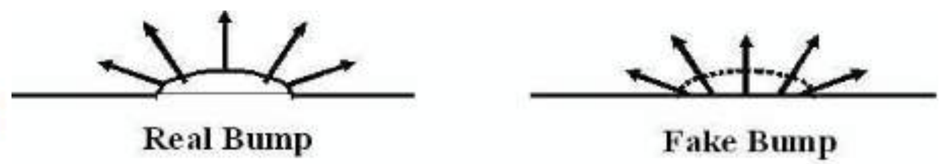
Environment mapping

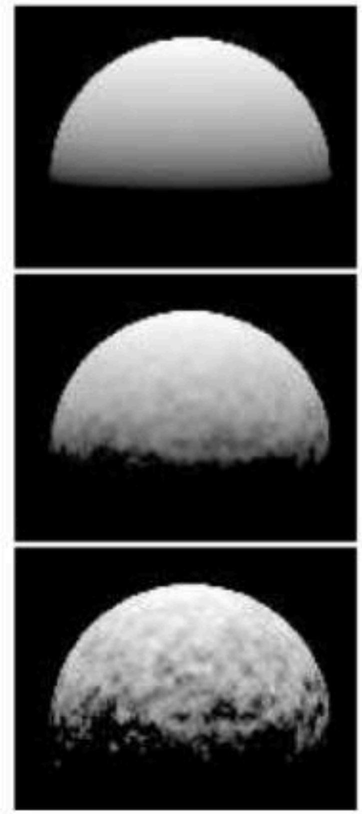
Image from slides by

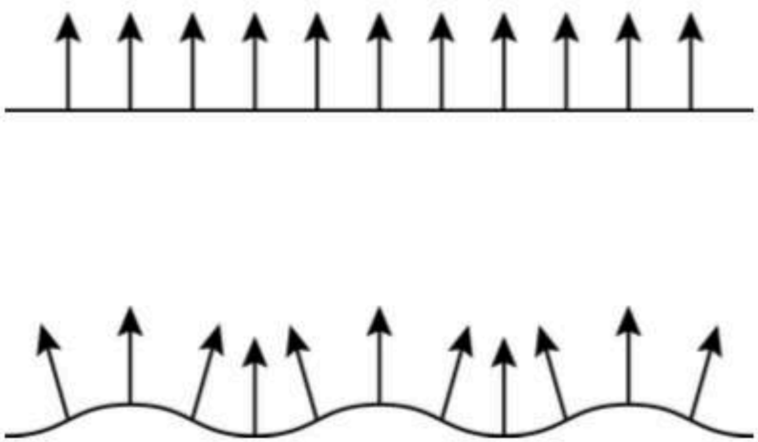
Bump mapping

One of the reasons why we apply texture mapping:

Real surfaces are hardly flat but often rough and bumpy. These bumps cause (slightly) different reflections of the light.

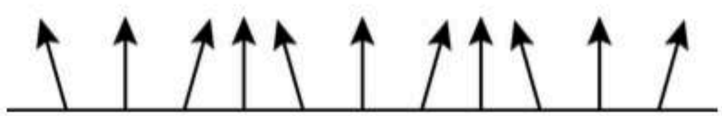


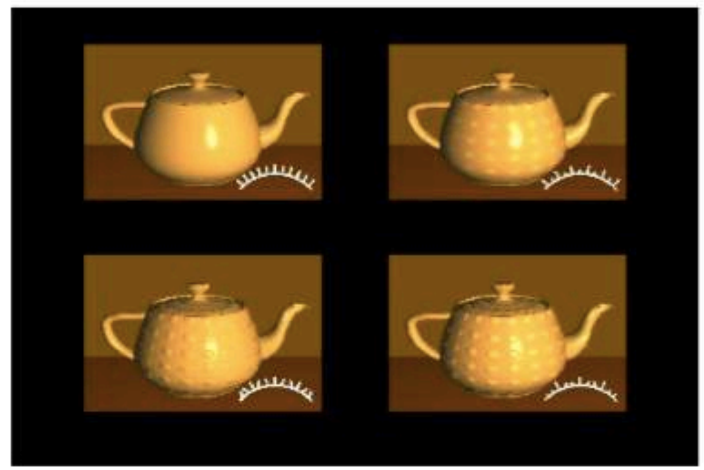




Bump mapping

Instead of mapping an image or noise onto an object, we can also apply a bump map, which is a 2D or 3D array of vectors. These vectors are added to the normals at the points for which we do shading calculations.

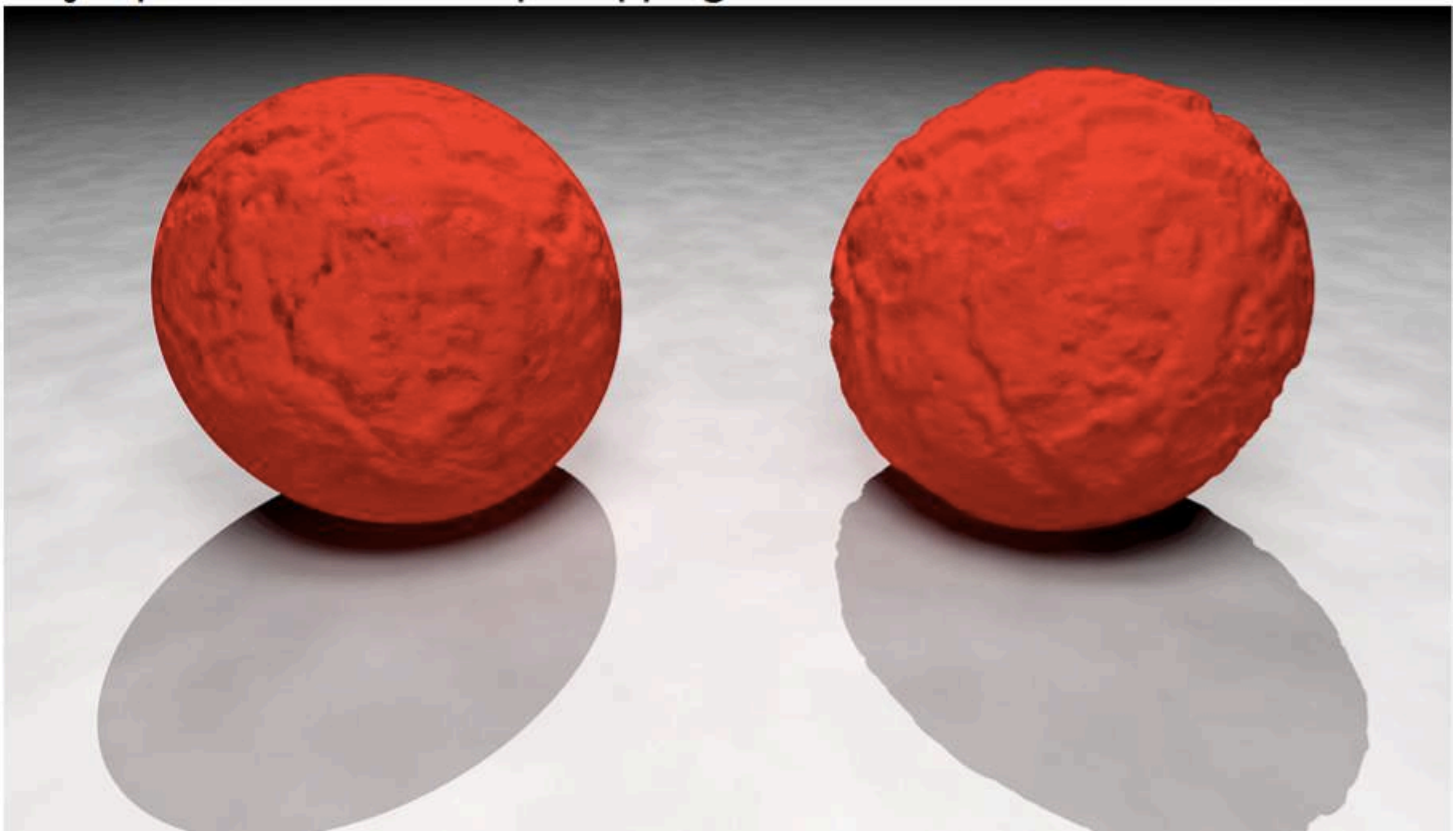




The effect of bump mapping is an apparent change of the geometry of the object.

Bump mapping

Major problems with bump mapping: silhouettes and shadows

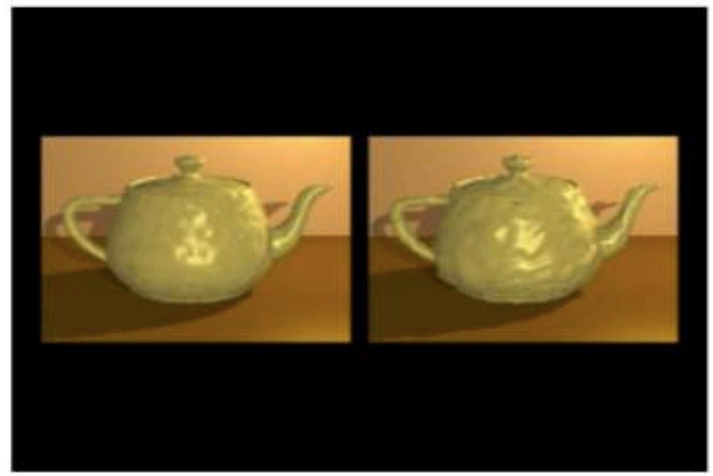



2D Image Bump Mapping Using a 24-bit Bitmap

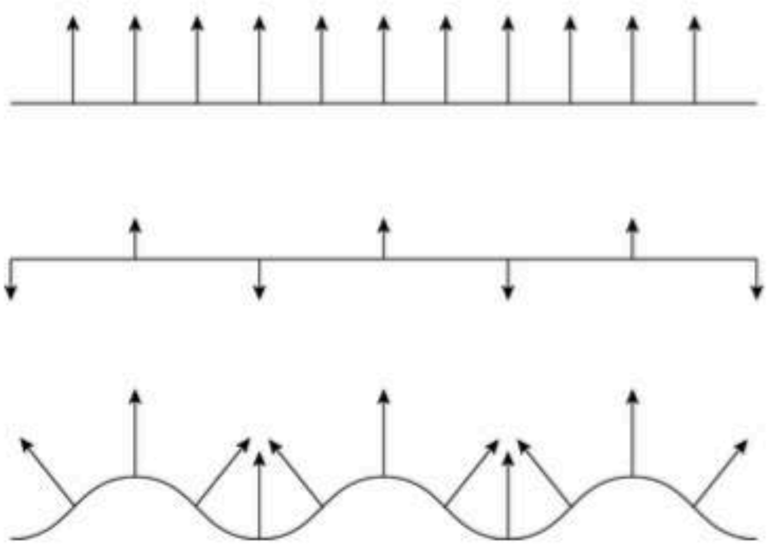
Displacement mapping

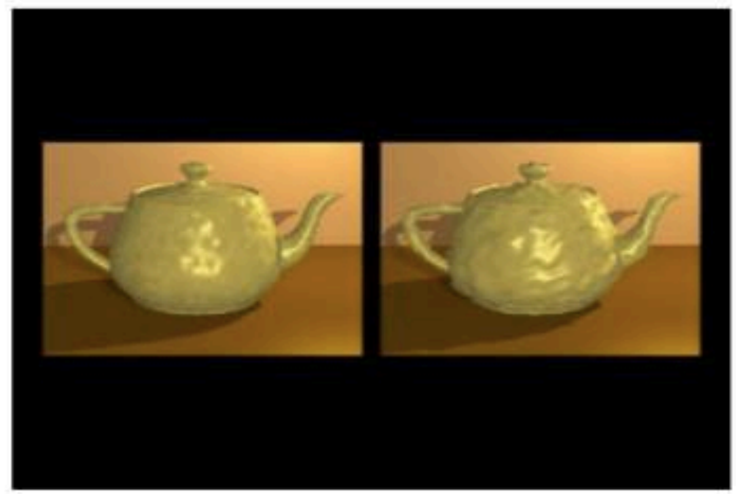
To overcome this shortcoming, we can use a displacement map. This is also a 2D or 3D array of vectors, but here the points to be shaded are actually displaced.

Normally, the objects are refined using the displacement map, giving an increase in storage requirements.



Displacement mapping





Topic 2:

Basic Ray Tracing

- Introduction to ray tracing
- Computing rays
- Computing intersections
    - ray-triangle
    - ray-polygon
    - ray-quadric
    - the scene signature
- Computing normals
- Evaluating shading model
- Spawning rays
- Incorporating transmission
    - refraction
    - ray-spawning & refraction

Local vs. Global Illumination

Local Illumination Models

e.g. Phong

- Model source from a light reflected once off a surface towards the eye
- Indirect light is included with an ad hoc “ambient” term which is normally constant across the scene

Global Illumination Models

e.g. ray tracing or radiosity (both are incomplete)

- Try to measure light propagation in the scene
- Model interaction between objects and other objects, objects and their environment

All surfaces are not created equal

Specular surfaces

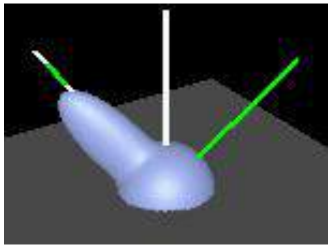
- e.g. mirrors, glass balls
- An idealized model provides ‘perfect’ reflection

  Incident ray is reflected back as a ray in a single direction

Diffuse surfaces

- e.g. flat paint, chalk
- Lambertian surfaces
- Incident light is scattered equally in all directions

General reflectance model: **BRDF**



Categories of light transport

Specular-Specular

Specular-Diffuse

Diffuse-Diffuse

Diffuse-Specular

Ray Tracing

Traces path of specularly reflected or transmitted (refracted) rays through environment

Rays are infinitely thin

Don’t disperse

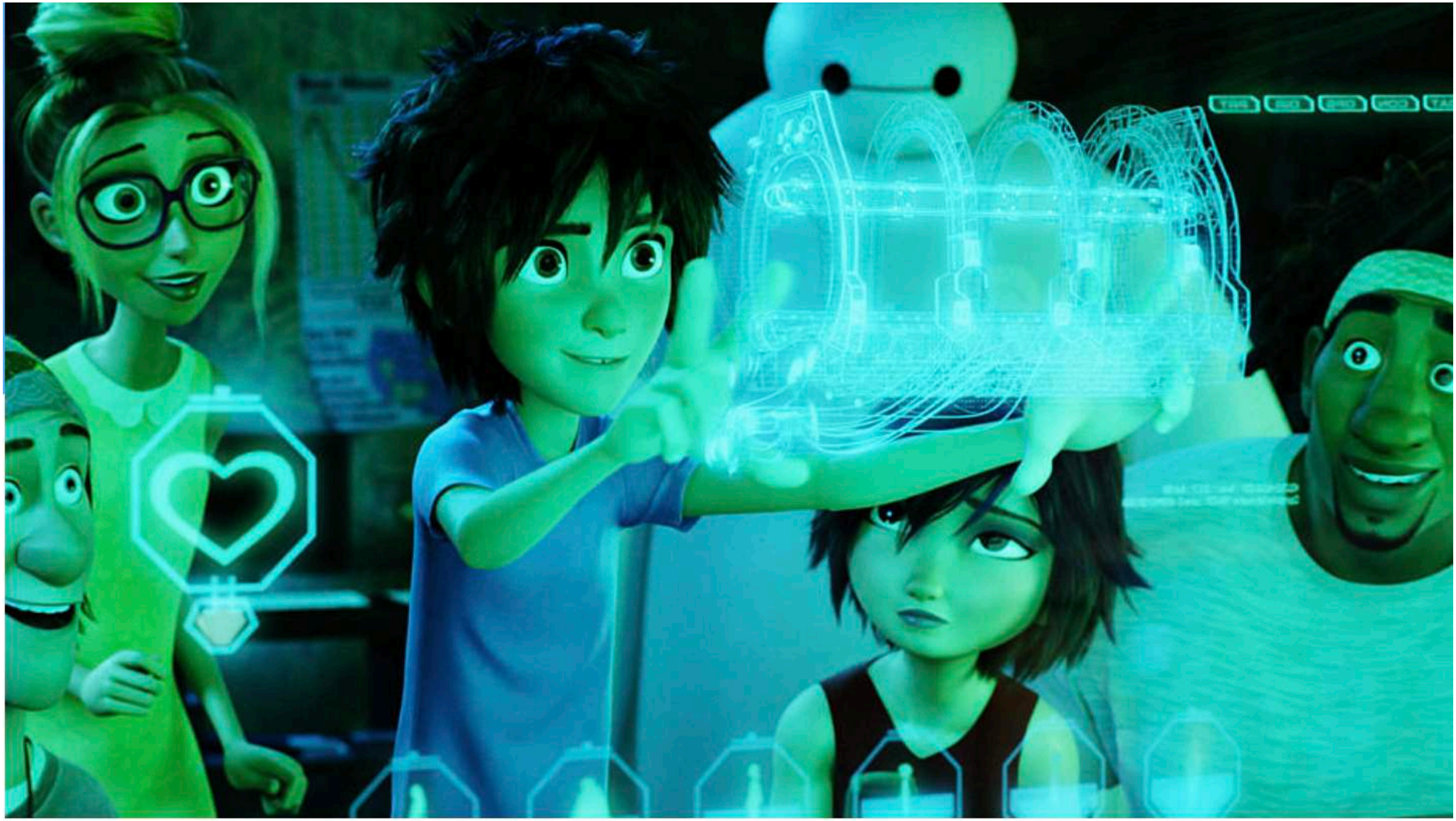
Signature: shiny objects exhibiting sharp, multiple reflections

Transport E - S – S – S – D – L.

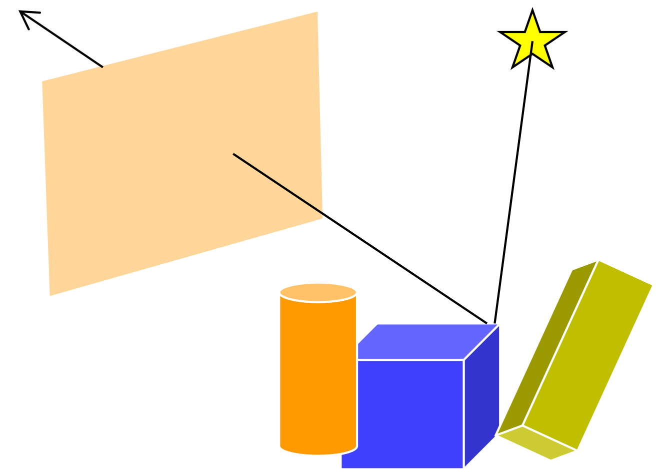
Ray Tracing

Unifies in one framework

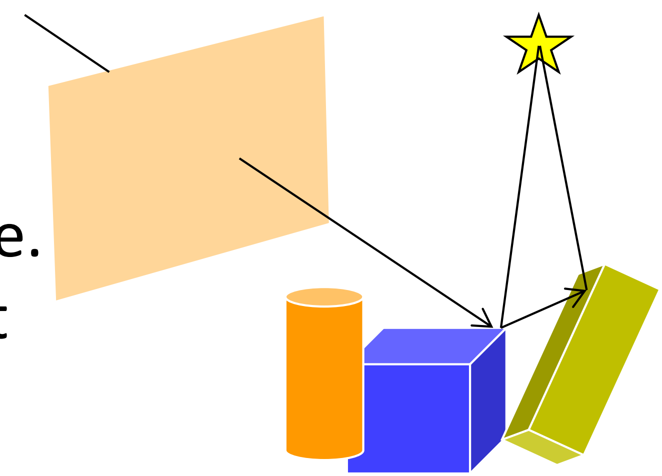
- Hidden surface removal
- Shadow computation
- Reflection of light
- Refraction of light
- Global **specular** interaction



Rasterization vs. Ray Tracing



Rasterization:
-project geometry onto image.
-pixel color computed by local illumination
(direct lighting).



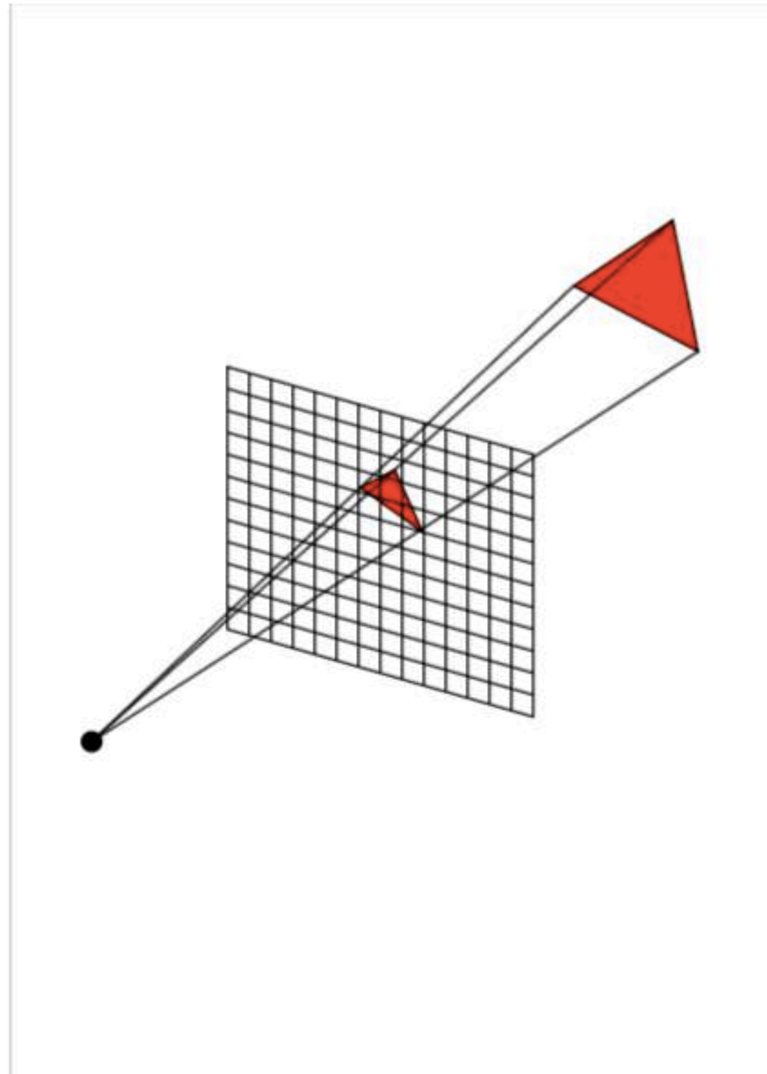
Ray-Tracing:
-project image pixels (backwards) onto scene.
-pixel color determined based on direct light
as well indirectly by recursively following
promising lights path of the ray.

Projective methods

A popular method for generating images from a 3D-model is projection, e.g.:

- 3D triangles project to 2D triangles
- Project vertices
- Fill/shade 2D triangle

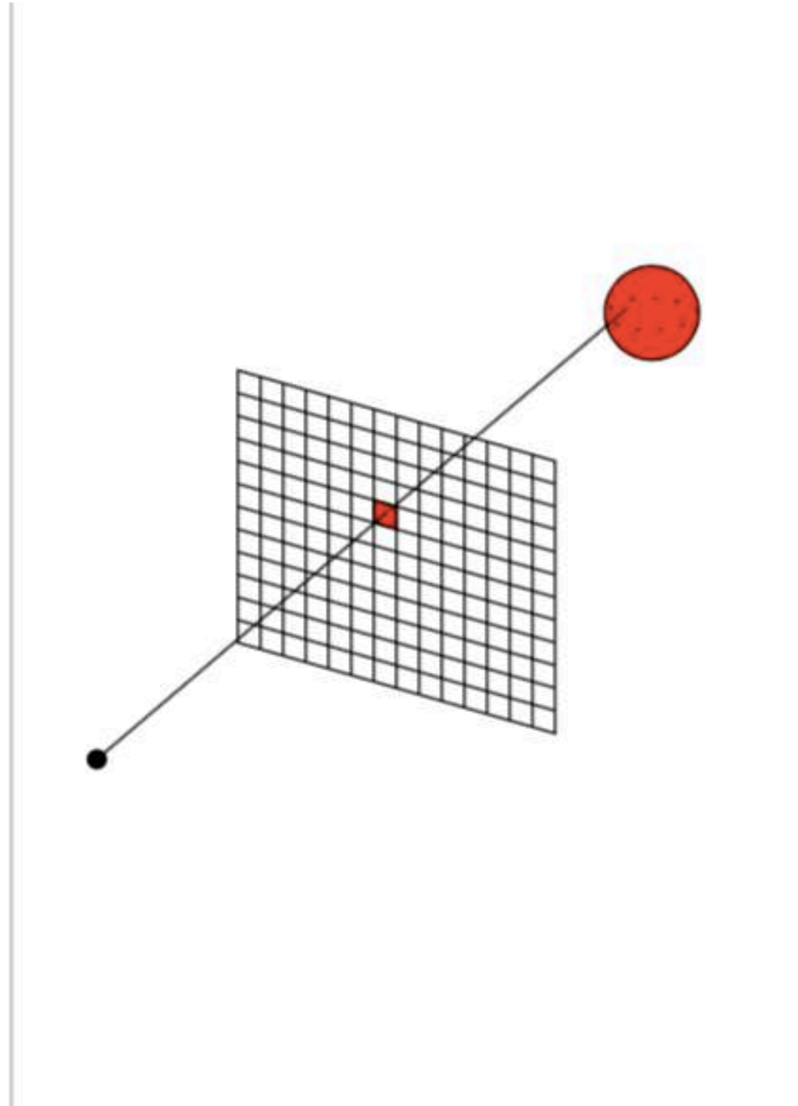
Notice:
Ray tracing = pixel-based,
proj. methods = object-based



Ray tracing / ray casting

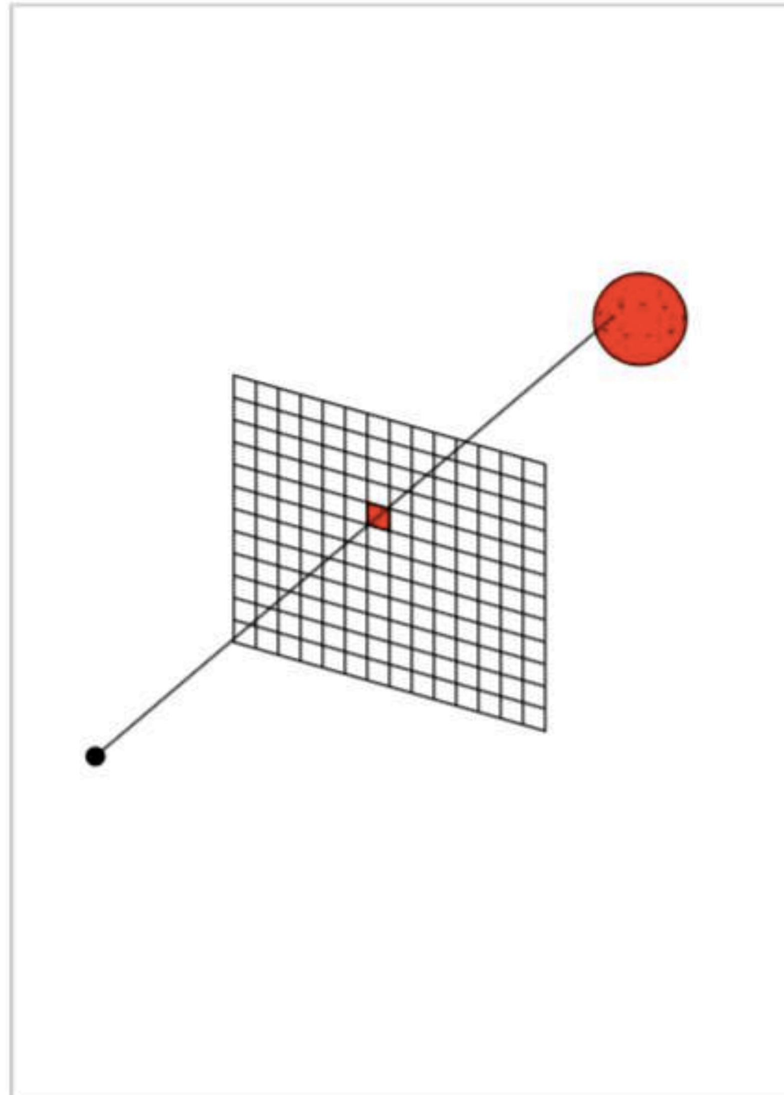
For photo-realistic rendering, usually ray tracing algorithms are used: for every pixel

- Compute ray from viewpoint through pixel center
- Determine intersection point with first object hit by ray
- Calculate shading for the pixel (possibly with recursion)



Ray tracing / ray casting

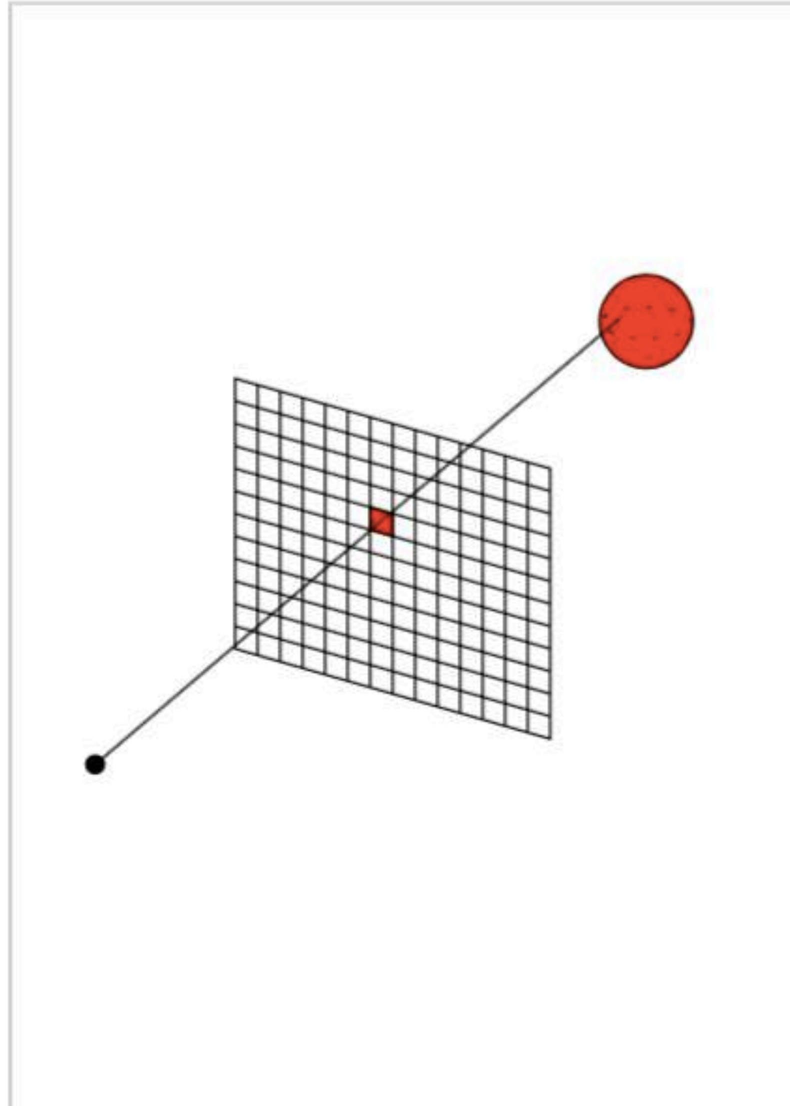
- Global Illumination
- Traditionally (very) slow
- Recent developments: real-time ray tracing



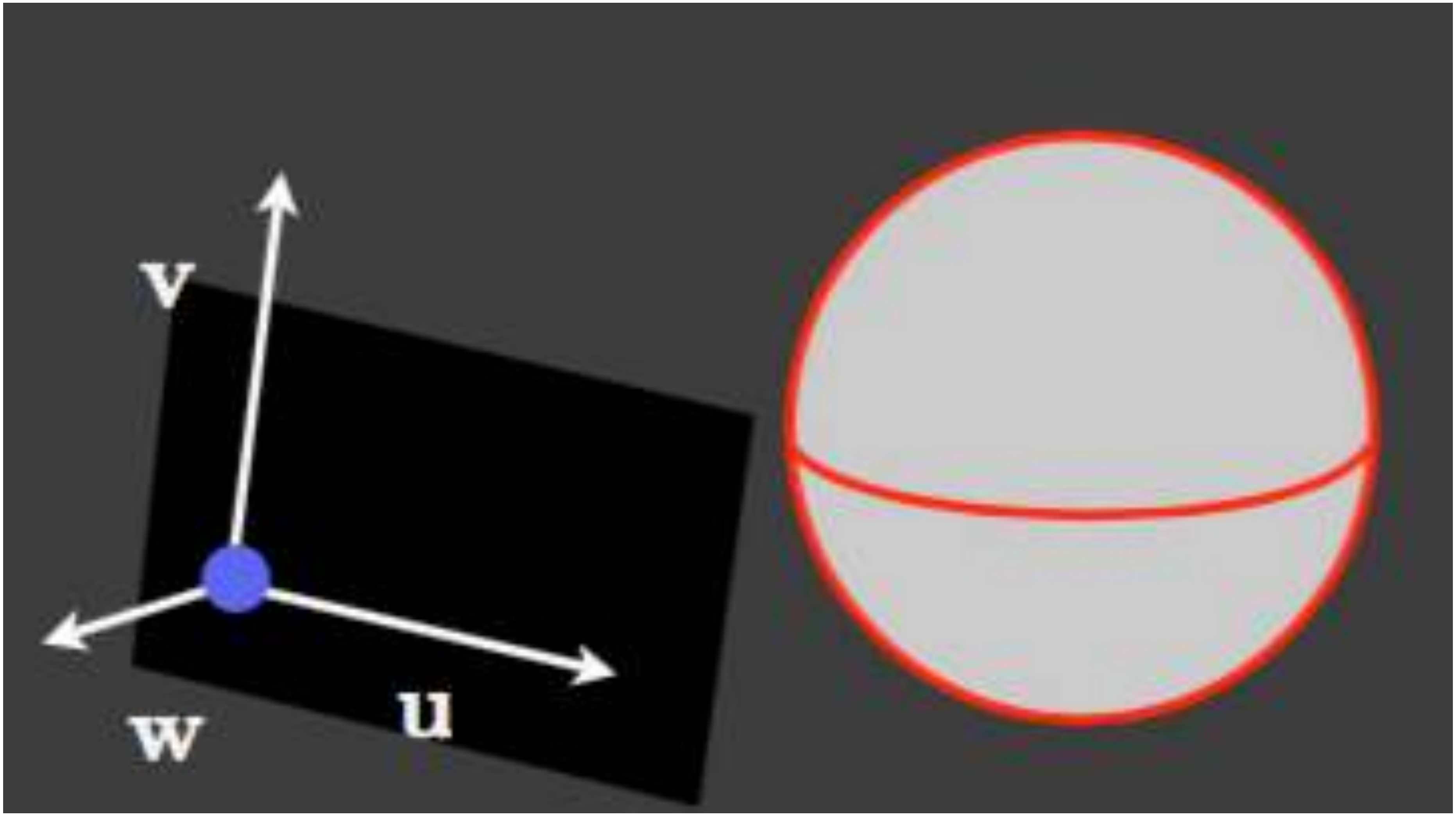
Ray tracing / ray casting

Why ray tracing is important (even if you are just interested in real-time rendering):

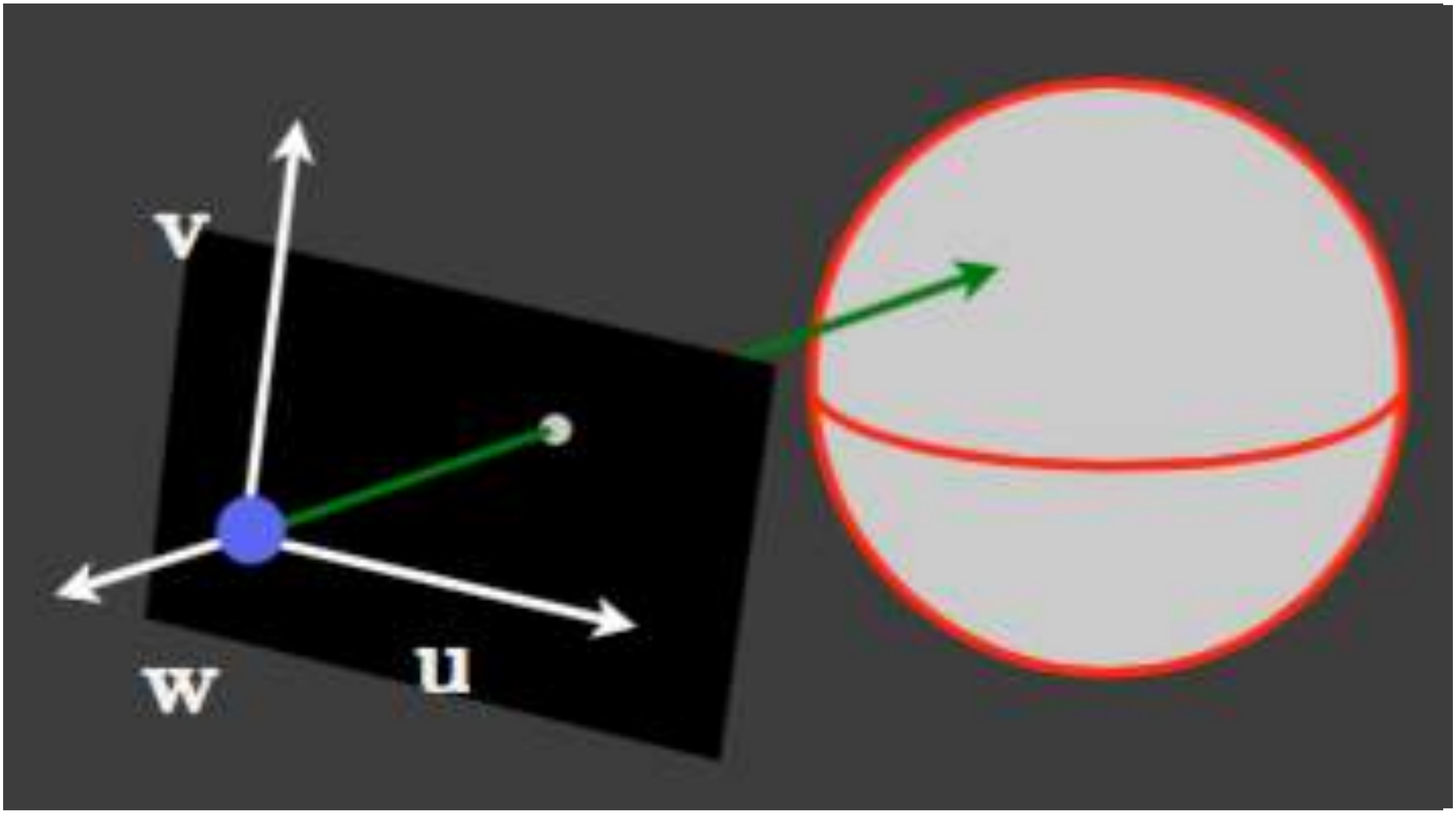
- Recent developments: real-time ray tracing, path tracing, etc.
- Important in games for interaction
- Important computer graphics technique (also: shares many techniques with other approaches)



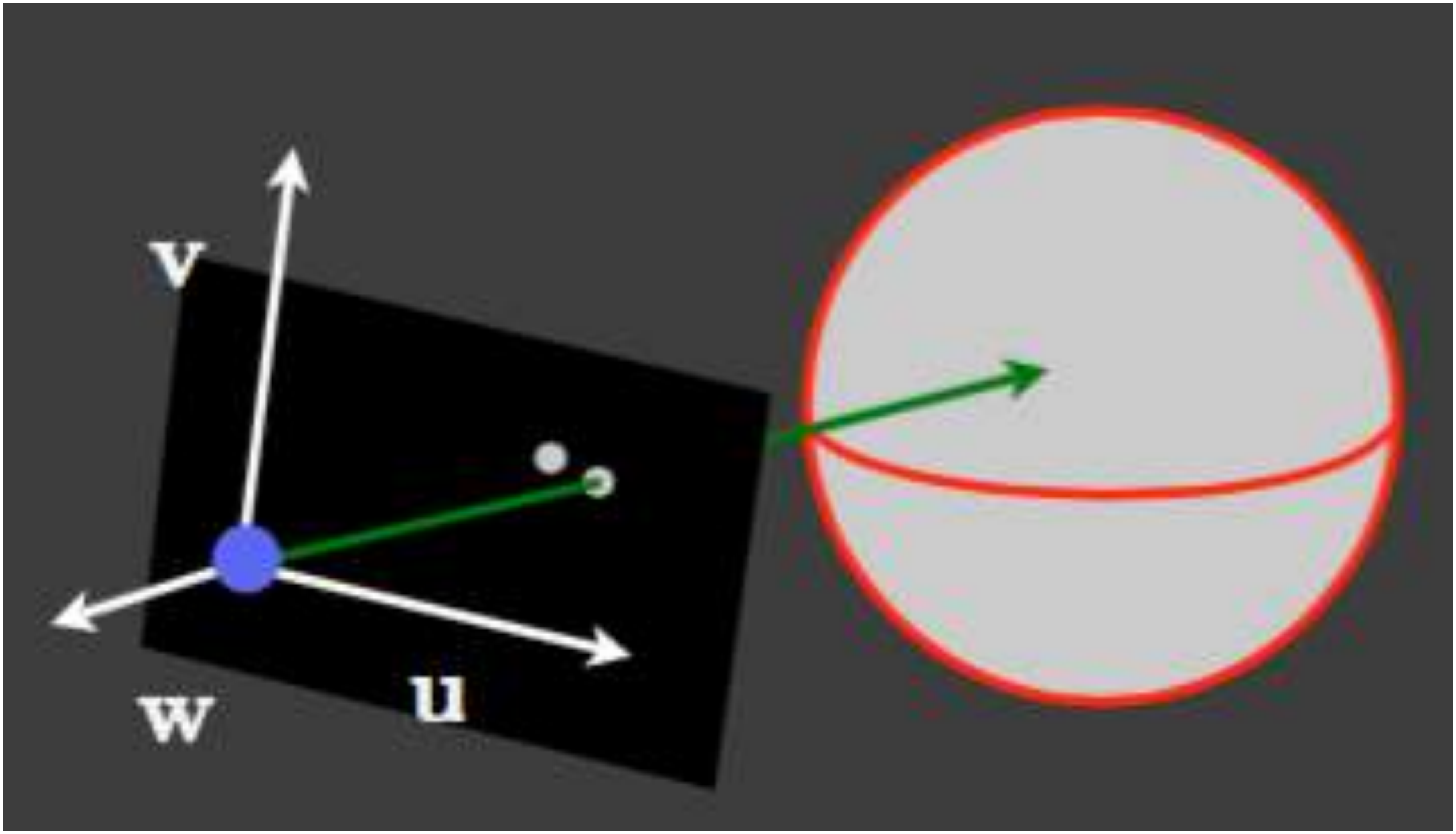
Ray tracing



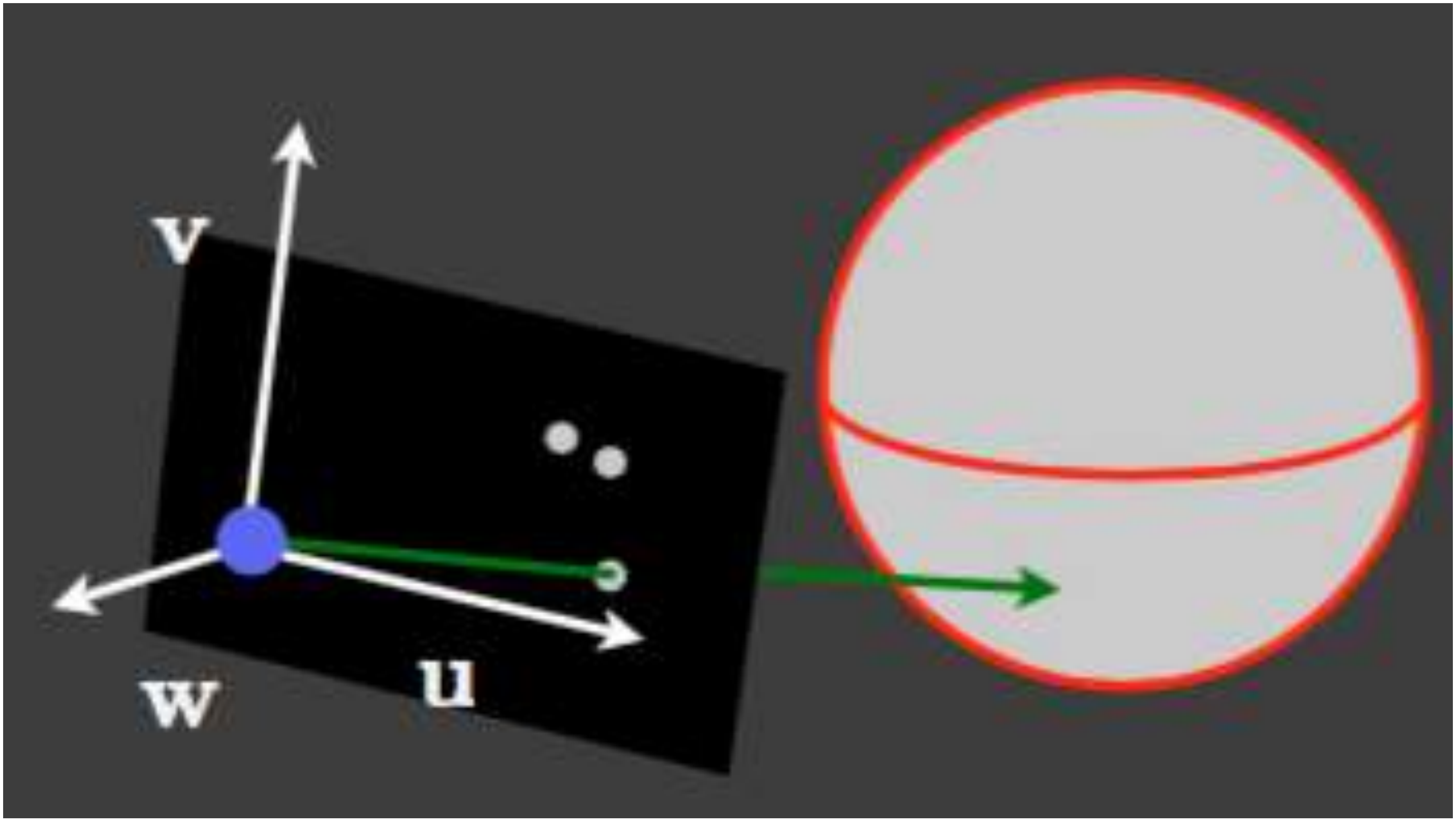
Ray tracing



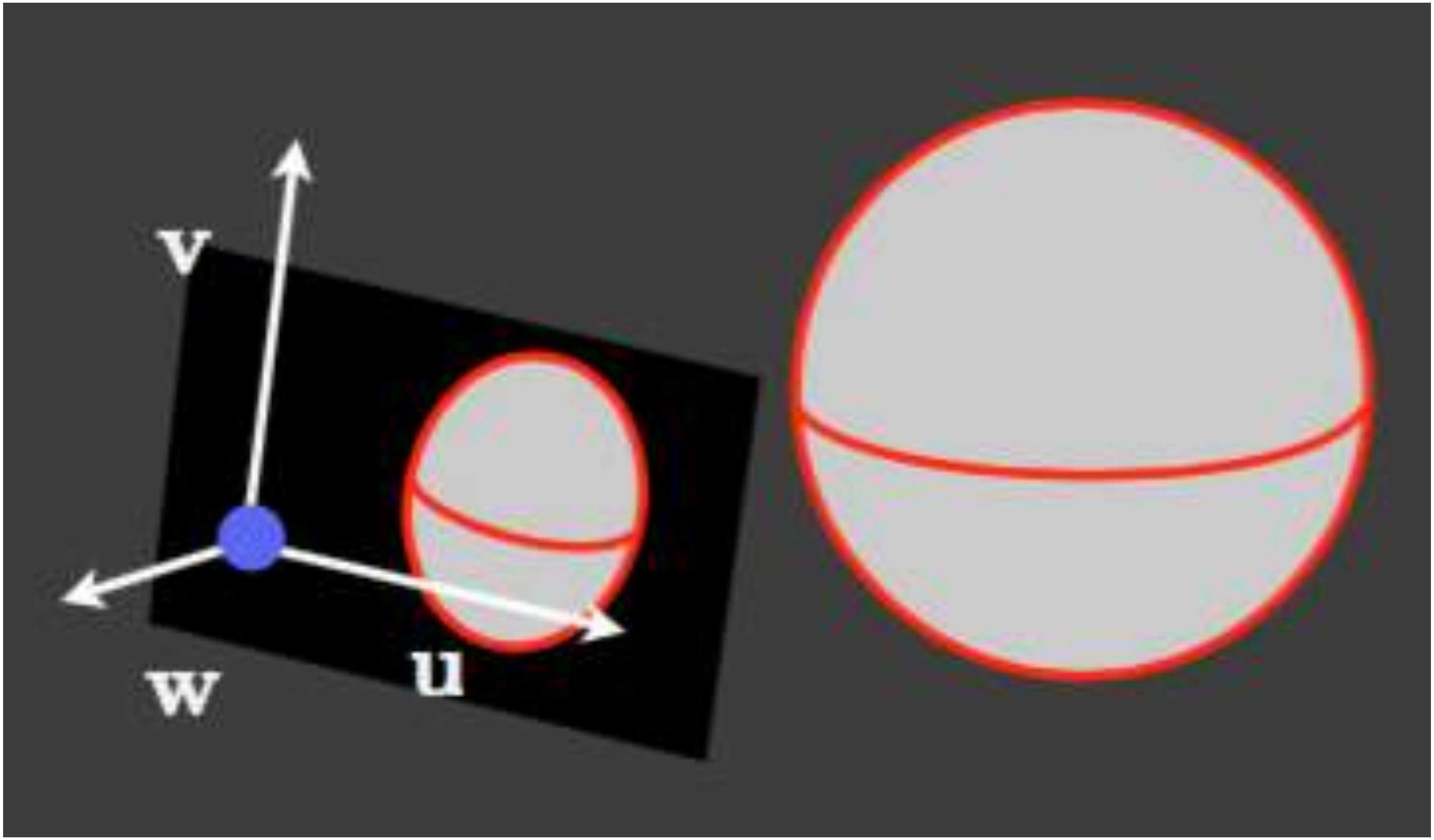
Ray tracing



Ray tracing



Ray tracing



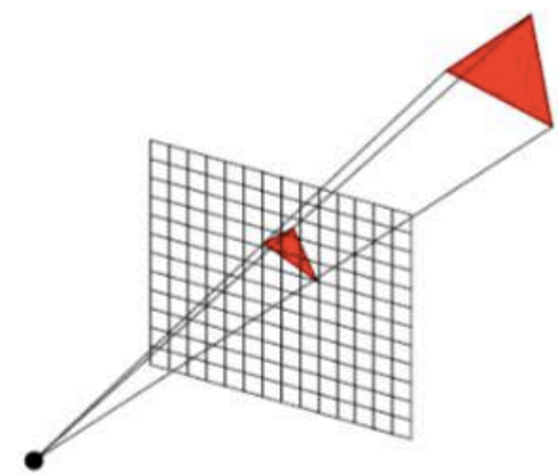
Projective methods vs. ray tracing

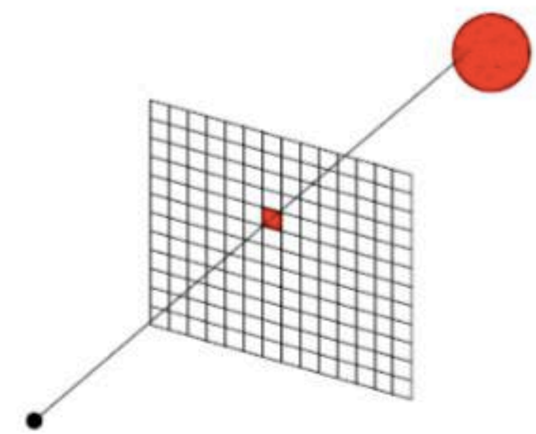
Projective methods & Ray tracing

. . . share lots of techniques,
e.g., shading models,
calculation of intersections, etc.

. . . but also have major differences,
e.g., projection and hidden
surface removal come "for free"
in ray tracing

And most importantly . . .



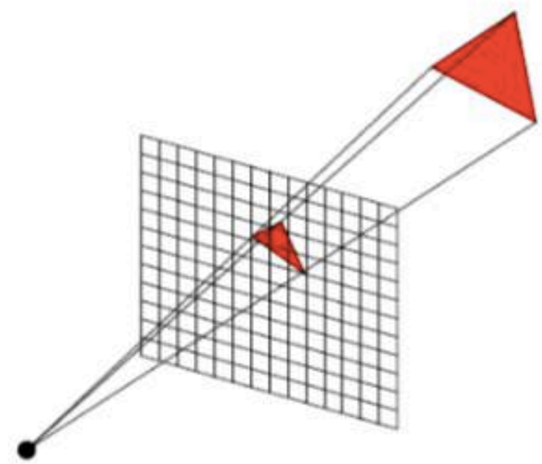


Projective methods vs. ray tracing

Projective methods:

Object-order rendering, i.e.

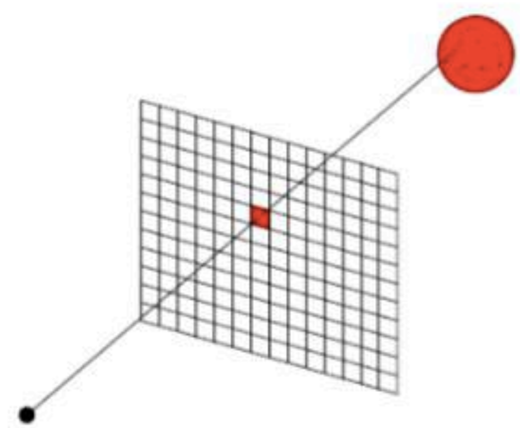
- For each object . . .
- . . . find and update all pixels that it influences and draw them accordingly



Ray tracing:

Image-order rendering, i.e.

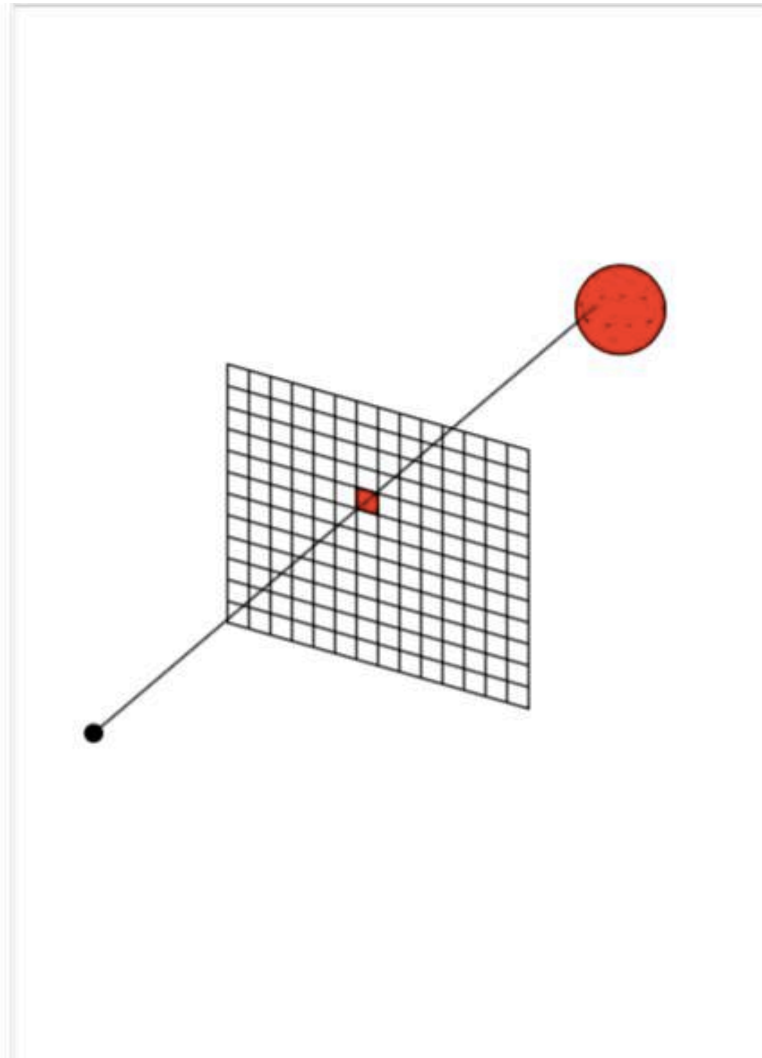
- For each pixel . . .
- . . . find all objects that influence it and update it accordingly



A basic ray tracing algorithm

FOR each pixel DO

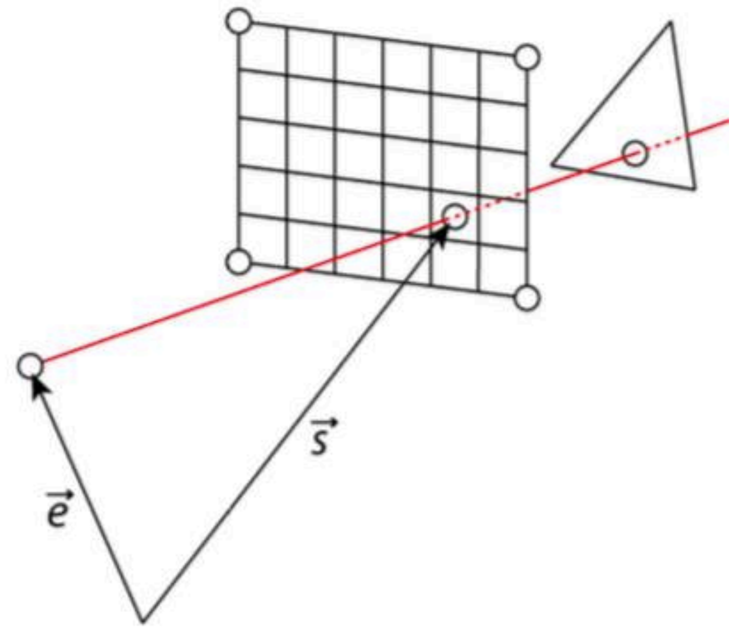
- compute viewing ray
- find the 1st object hit
  by the ray
  and its surface normal $\vec{n}$
- set pixel color to value
  computed from hit point,
  light, and $\vec{n}$



Lines and rays

We need to "shoot" a ray

- from the view point $\vec{e}$
- through a pixel $\vec{s}$ on the screen
- towards the scene/objects



Hmm, that should be easy with . . .

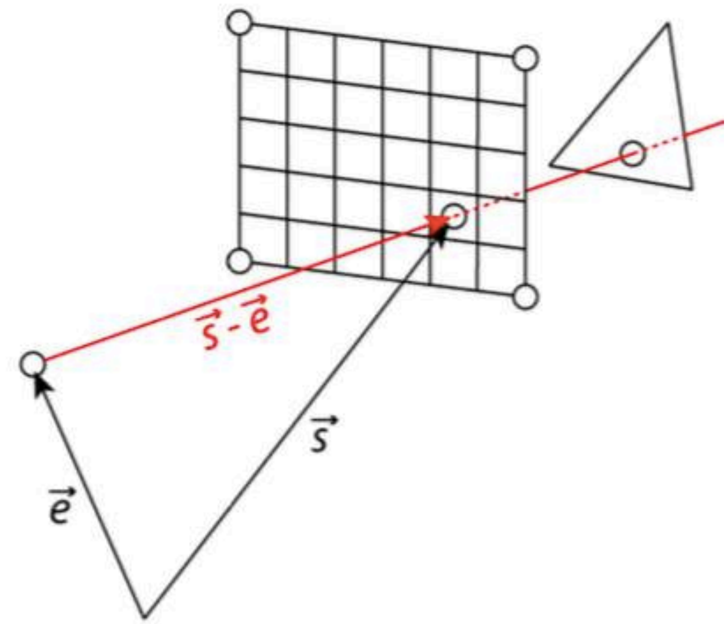
Lines and rays

. . . a parametric line equation:

$$\vec{p}(t) = \vec{e} + t(\vec{s} - \vec{e})$$

where

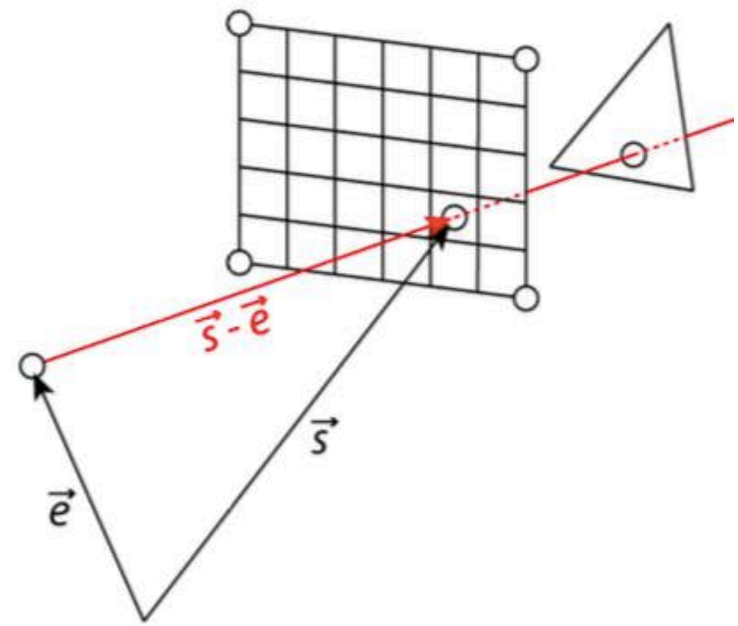
- $\vec{e}$ is a point on the line (aka its *support vector*)
- $\vec{s} - \vec{e}$ is a vector on the line (aka its *direction vector*)



Lines and rays

With this, our ray ...

- starts at $\vec{e}$ $(t = 0)$,
- goes throught $\vec{s}$ $(t = 1)$,
- and "shoots" towards the scene/objects $(t > 1)$



Hmm, calculation would become much easier if we would have ...

Coordinate system

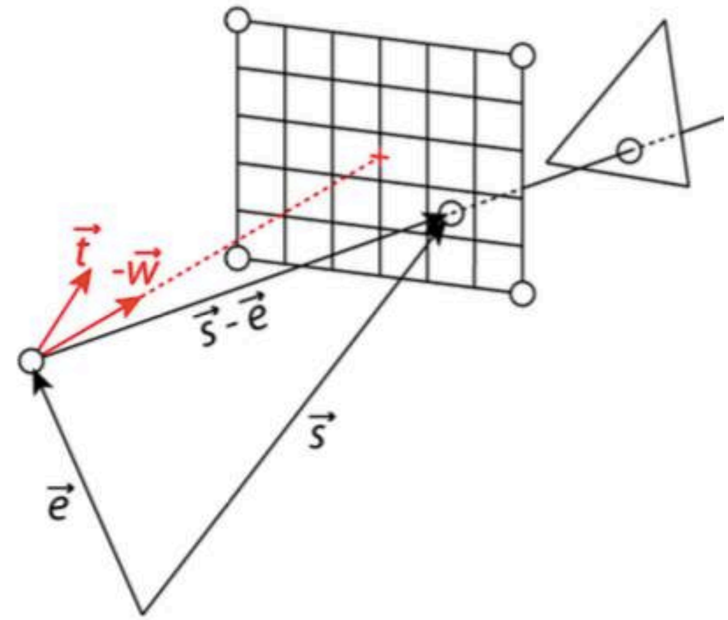
... a camera coordinate system:

That's easy! Using

- our camera position $\vec{e}$
- our viewing direction $-\vec{w}$
- and a view up vector $\vec{t}$

we get

- $\vec{u} = -\vec{w} \times \vec{t}$
- $\vec{v} = -\vec{w} \times \vec{u}$



Coordinate system

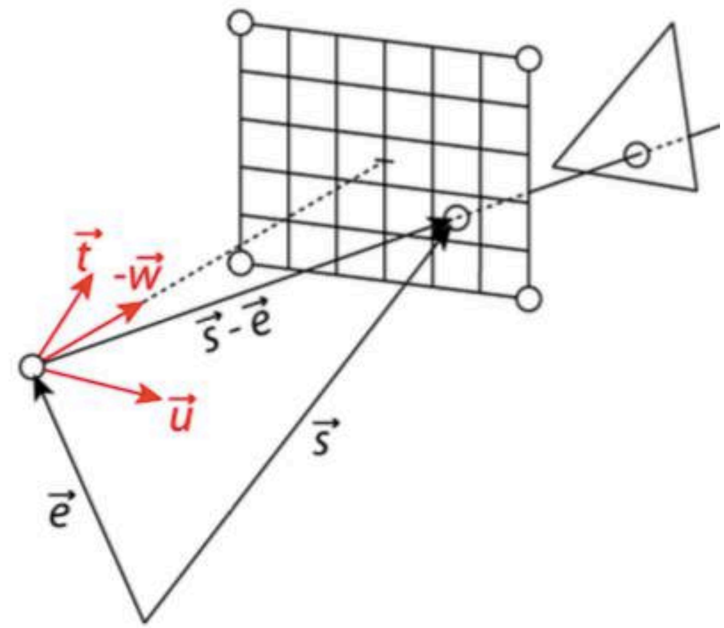
. . . a camera coordinate system:

That's easy! Using

- our camera position $\vec{e}$
- our viewing direction $-\vec{w}$
- and a view up vector $\vec{t}$

we get

- $\vec{u} = -\vec{w} \times \vec{t}$
- $\vec{v} = -\vec{w} \times \vec{u}$



Coordinate system

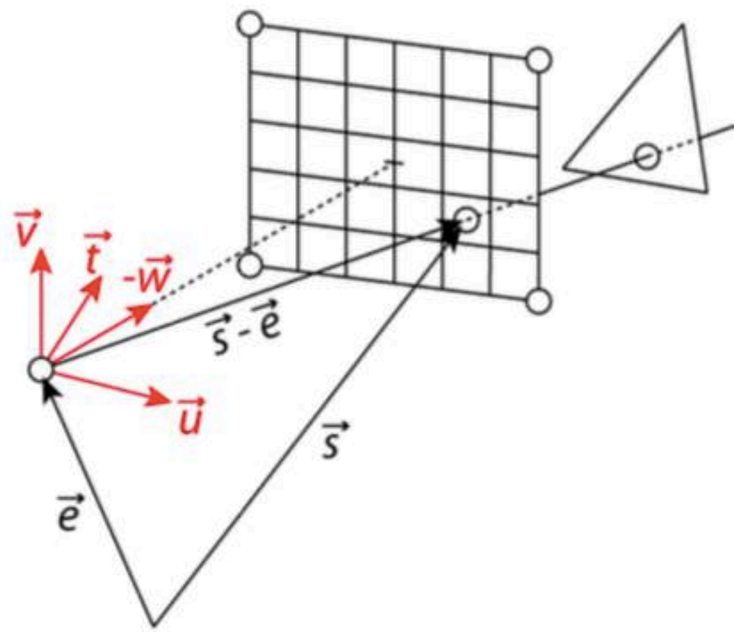
... a camera coordinate system:

That's easy! Using

- our camera position $\vec{e}$
- our viewing direction $-\vec{w}$
- and a view up vector $\vec{t}$

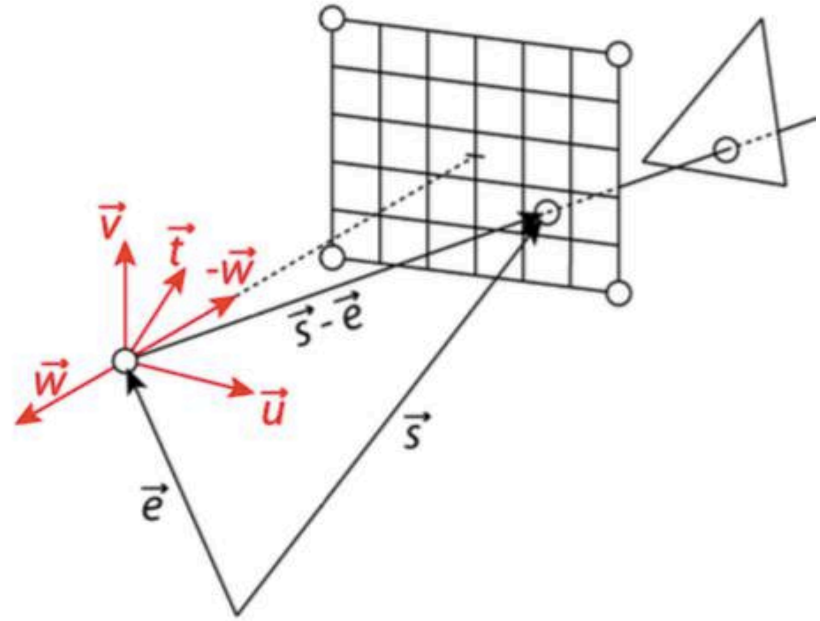
we get

- $\vec{u} = -\vec{w} \times \vec{t}$
- $\vec{v} = -\vec{w} \times \vec{u}$



Coordinate system

Notice that we chose $-\vec{w}$ as viewing direction and not $\vec{w}$, in order to get a right handed coordinate system.

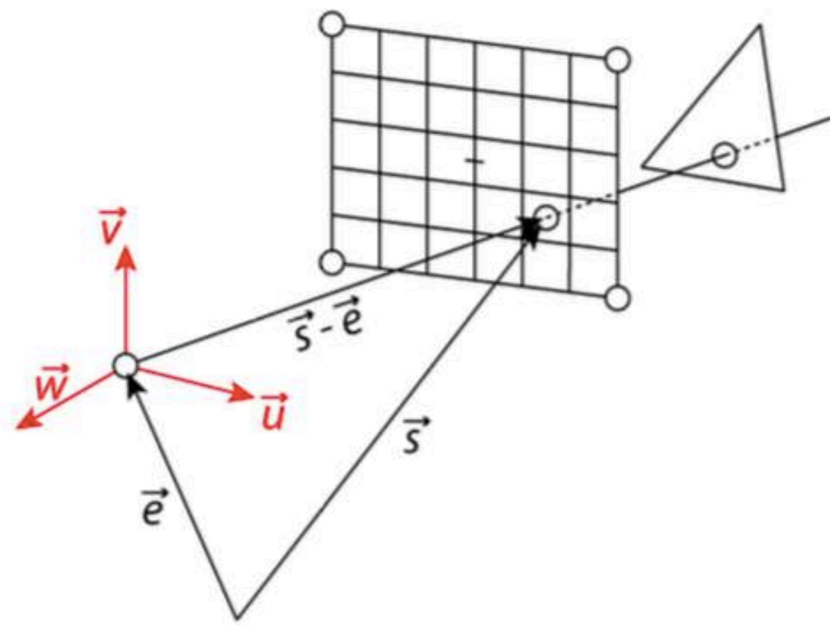


Coordinate system

Normalizing, i.e.

- $\vec{w}/||\vec{w}||$
- $\vec{u}/||\vec{u}||$
- $\vec{v}/||\vec{v}||$

gives us our coordinate system.



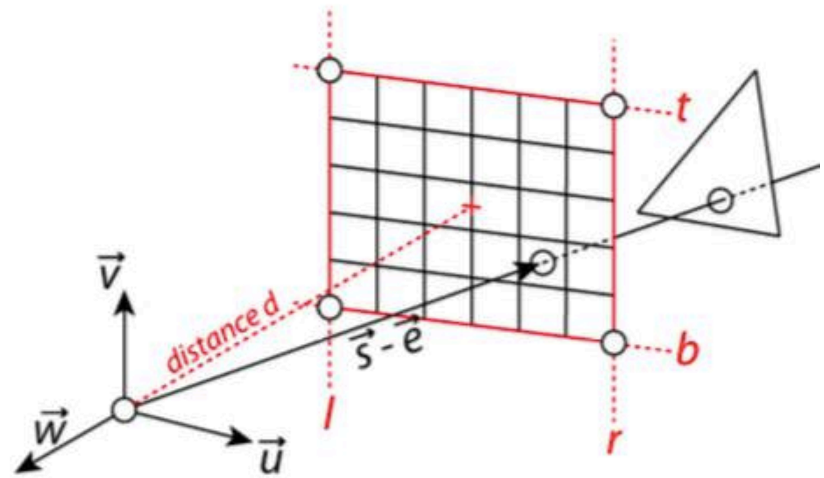
Viewing window

With this new coordinate system we can easily define our viewing window:

- left side: $u = l$
- right side: $u = r$
- top: $v = t$
- bottom: $v = b$

Plus the viewing plane at a distance d from the eye/camera:

- distance: $-w = d$

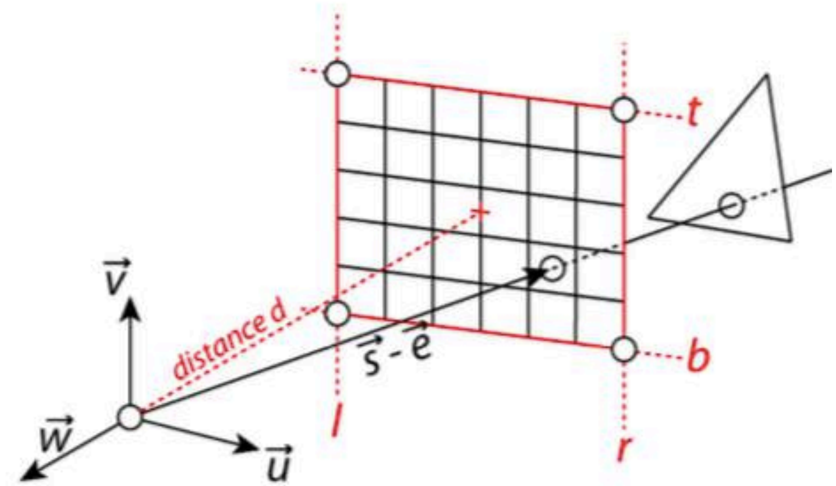


Viewing window

Assuming our window has $n_x \times n_y$ pixels, expressing a pixel position (i, j) on the viewing window in our new coordinate system (u, v) can be done with a simple window transformation from $n_x \times n_y$ to $(r - l) \times (t - b)$:

$$u = l + (r - l)(i + 0.5)/n_x$$

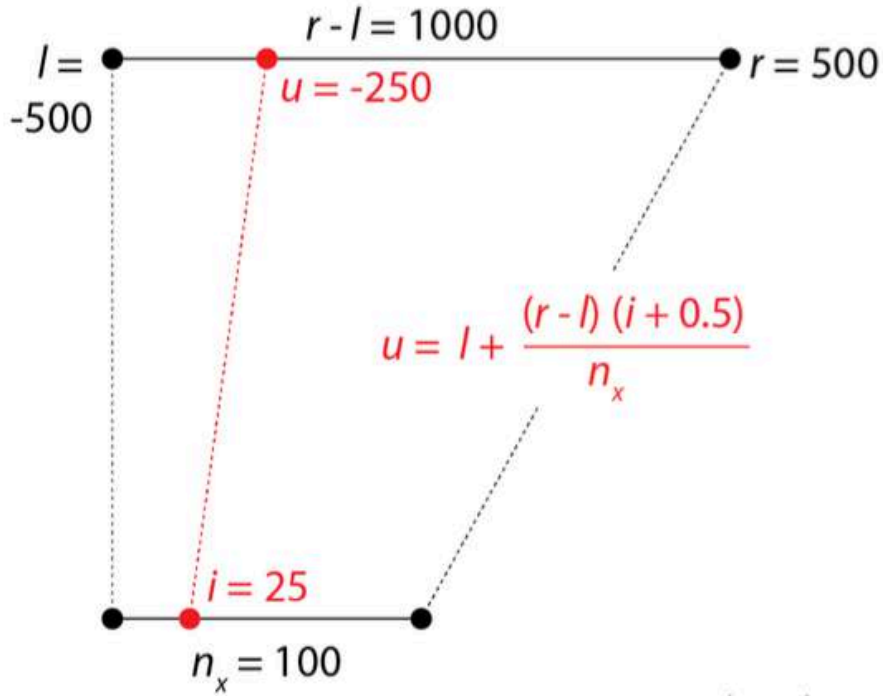
$$v = b + (t - b)(j + 0.5)/n_y$$

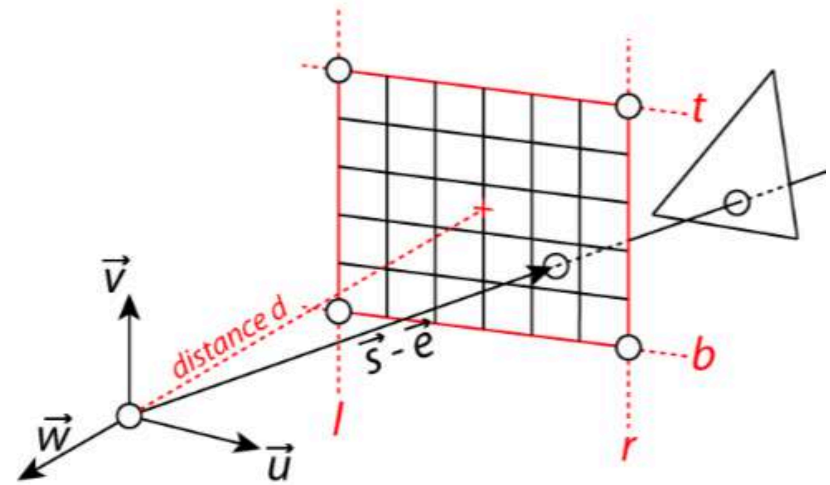


Viewing window

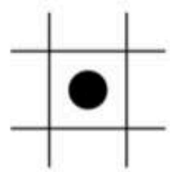
Example for u: Transformation from

$l = -500$, $r = 500$ to $n_x = 100$





Note: we add +0.5 to *i* because
we are dealing with pixel centers.



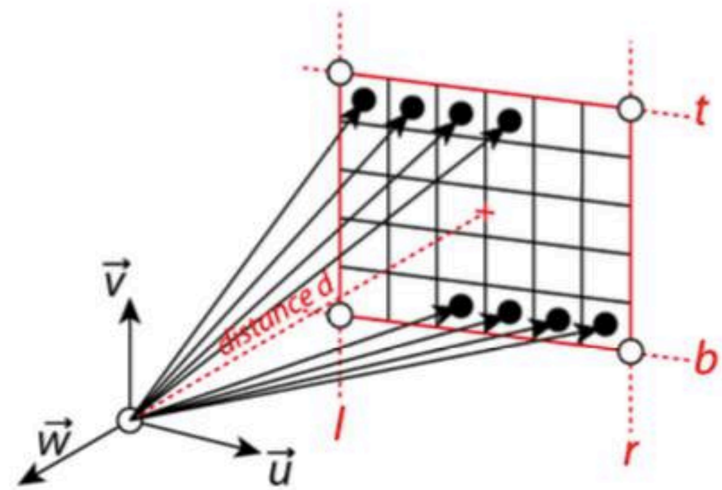
Viewing rays

For perspective views, viewing rays

- have the same origin $\vec{e}$
- but different direction

If d denotes the origin's distance to the plane, and u, v are calculated as before, we can write the direction as

- $u\vec{u} + v\vec{v} - d\vec{w}$.



Our viewing ray becomes

- $\vec{p}(\alpha) = \vec{e} + \alpha(u\vec{u} + v\vec{v} - d\vec{w})$

Viewing rays

For orthographic views, viewing rays

- have the same direction $-\vec{w}$
- but different origin

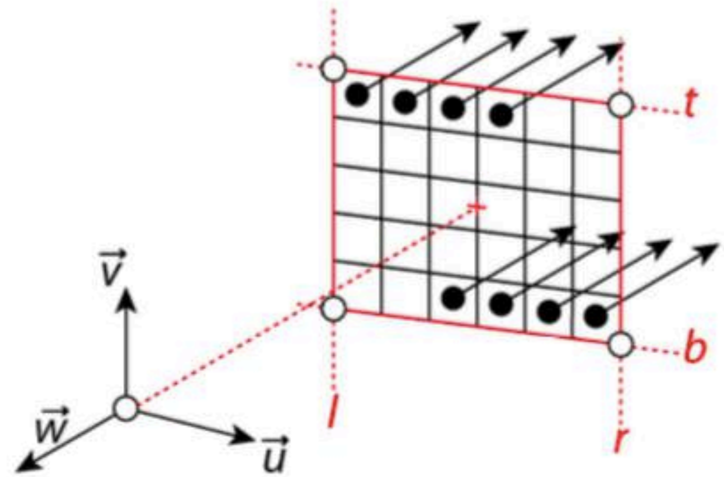
We get the origin with the previously introduced mapping from (i, j) to (u, v):

$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

and can write it as $\vec{e} + u\vec{u} + v\vec{v}$.

Our viewing ray becomes

- $\vec{p}(\alpha) = \vec{e} + u\vec{u} + v\vec{v} - \alpha\vec{w}$



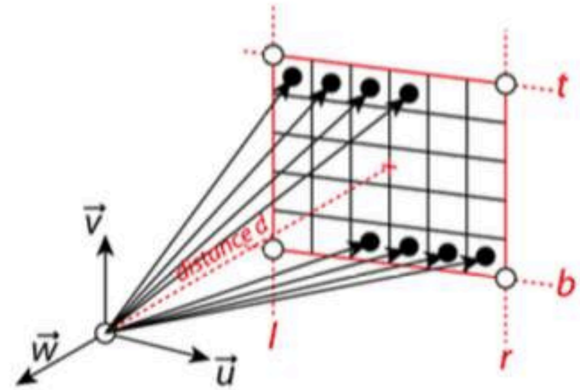
Viewing rays compared

Viewing rays for perspective views

- $\vec{p}(\alpha) = \vec{e} + \alpha(u\vec{u} + v\vec{v} - d\vec{w})$

with

- support vector $\vec{e}$
- direction vector $u\vec{u} + v\vec{v} - d\vec{w}$

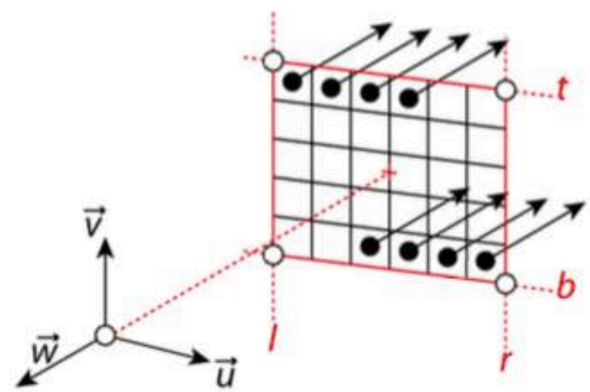


Viewing rays for orthographic views

- $\vec{p}(\alpha) = \vec{e} + u\vec{u} + v\vec{v} - \alpha\vec{w}$

with

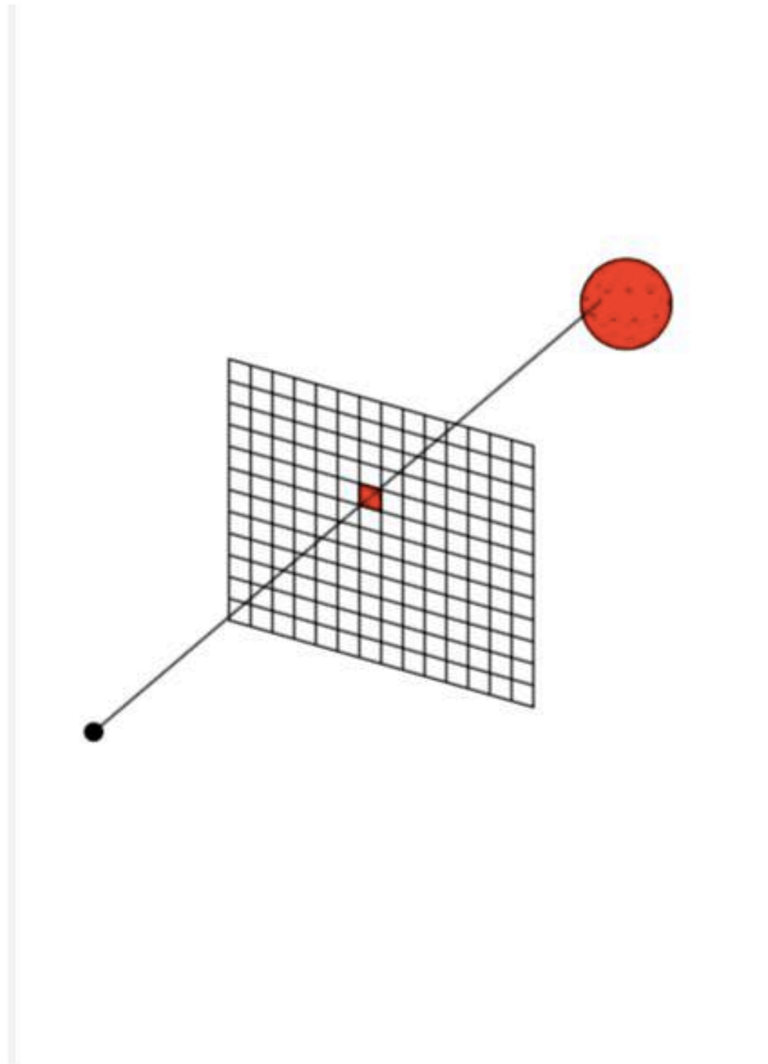
- support vector $\vec{e} + u\vec{u} + v\vec{v}$
- direction vector $-\vec{w}$



A basic ray tracing algorithm

FOR each pixel DO

- compute viewing ray
- find the 1st object hit by the ray and its surface normal $\vec{n}$
- set pixel color to value computed from hit point, light, and $\vec{n}$



Ray-object intersection (implicit surface)

In general, the intersection points of

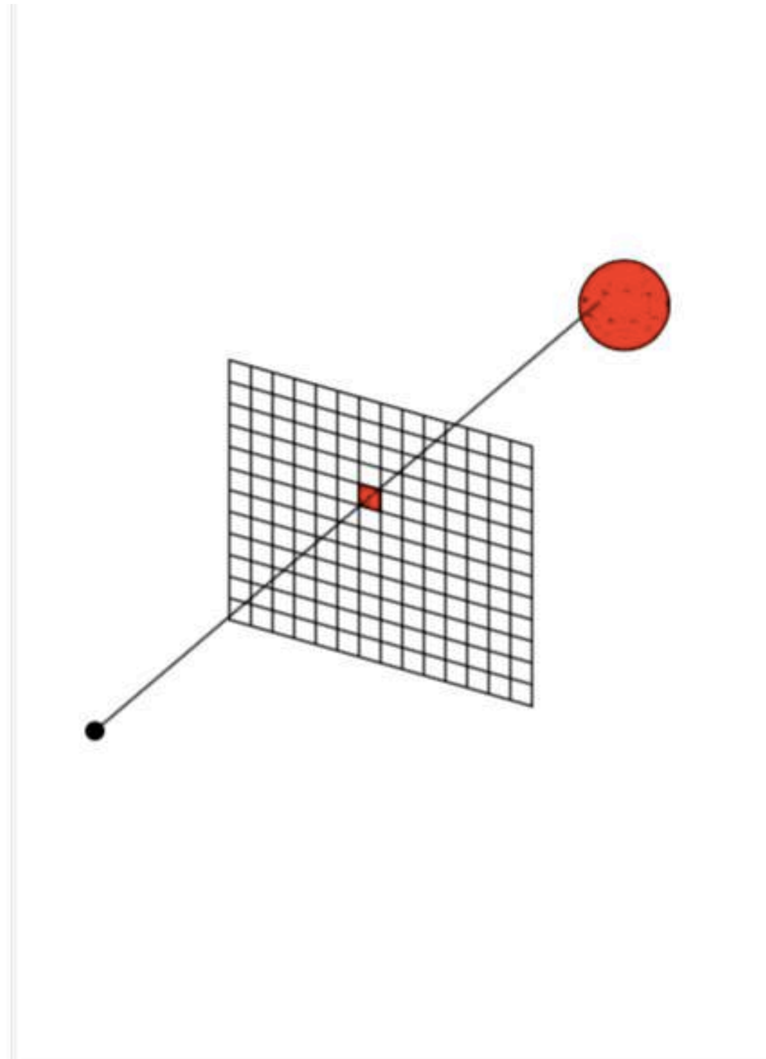
- a ray $\vec{p}(t) = \vec{e} + t\vec{d}$ and
- an implicit surface $f(\vec{p}) = 0$

can be calculated by

$$f(\vec{p}(t)) = 0$$

or

$$f(\vec{e} + t\vec{d}) = 0$$



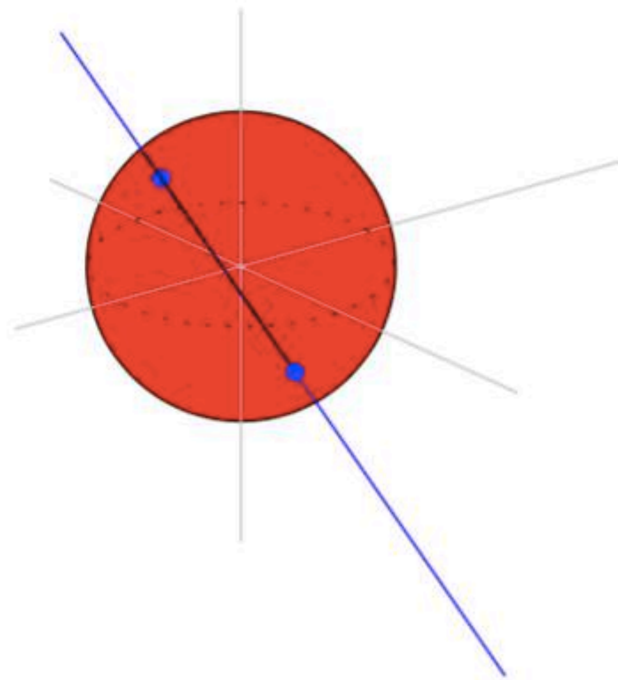
Spheres

The implicit equation for a sphere with center $\vec{c} = (x_c, y_c, z_c)$ and radius R is

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0$$

or in vector form

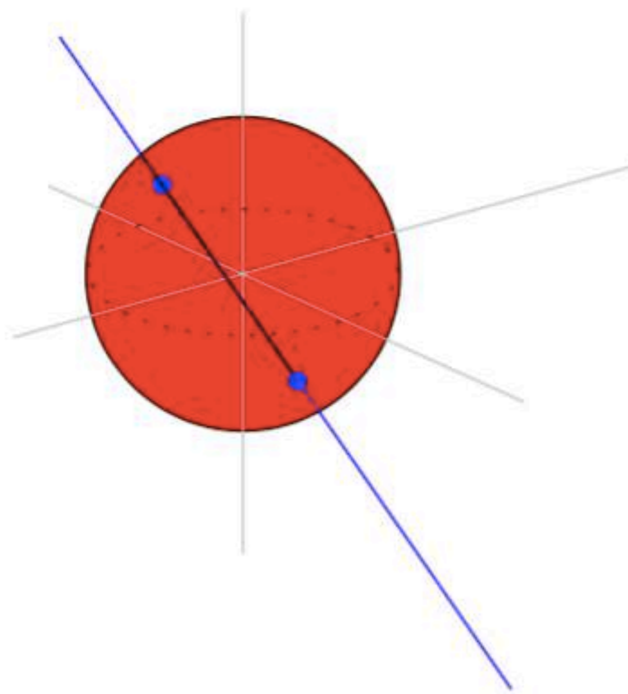
$$(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) - R^2 = 0$$



Intersections between rays and spheres

Intersection points have to fullfil

- the ray equation
  $\vec{p}(t) = \vec{e} + t\vec{d}$
- the sphere equation
  $(x - x_c)^2 + (y - y_c)^2$
  $+(z - z_c)^2 - R^2 = 0$



Hence, we get
$(\vec{e} + t\vec{d} - \vec{c}) \cdot (\vec{e} + t\vec{d} - \vec{c}) - R^2 = 0$

which is the same as
$(\vec{d}\cdot\vec{d})t^2 + 2\vec{d}\cdot(\vec{e} - \vec{c})t + (\vec{e} - \vec{c})\cdot(\vec{e} - \vec{c}) - R^2 = 0$

Intersections between rays and spheres

$$(\vec{d}\cdot\vec{d})t^2+2\vec{d}\cdot(\vec{e}-\vec{c})t+(\vec{e}-\vec{c})\cdot(\vec{e}-\vec{c})-R^2=0$$

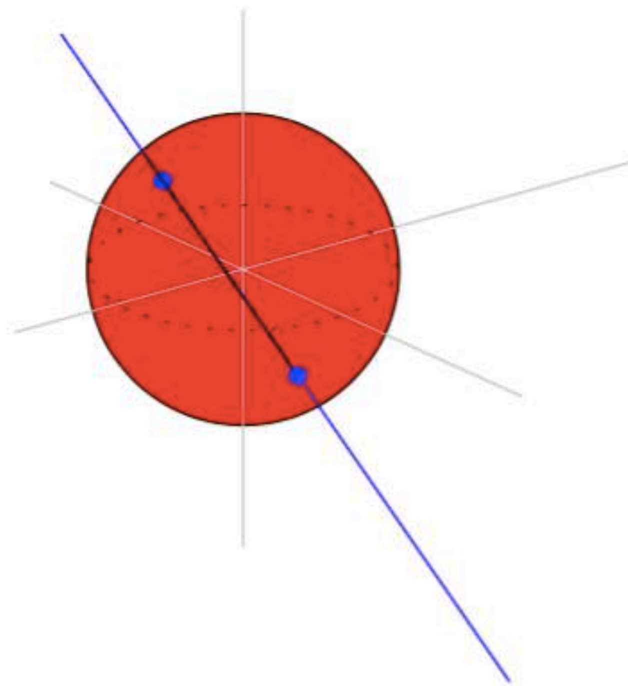
is a quadratic equation in t, i.e.

$$At^2+Bt+C=0$$

that can be solved by

$$t_{1,2}=\frac{-B\pm\sqrt{B^2-4AC}}{2A}$$

and can have 0, 1, or 2 solutions.



Intersections between rays and planes

Given a ray in parametric form, i.e.

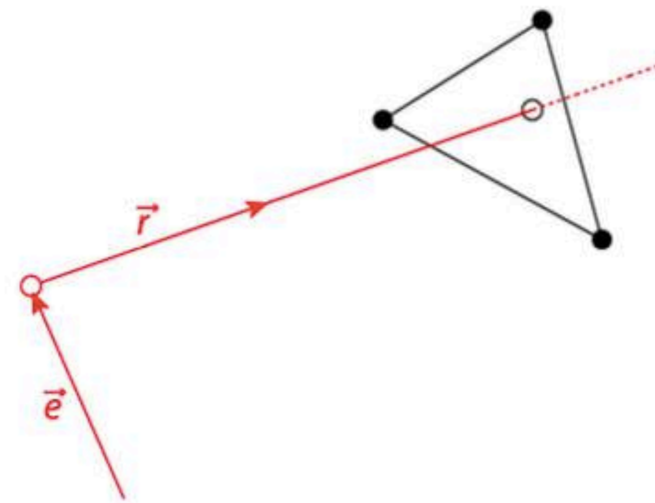
$$\vec{p}(t) = \vec{e} + t\vec{d}$$

and a plane in its implicit form, i.e.

$$(\vec{p} - \vec{p_1}) \cdot \vec{n} = 0$$

we can calculate the intersection point by putting the ray equation into the plane equation and solving for t, i.e.

$$t = \frac{(\vec{p_1} - e) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}$$



Ray-object intersection (parametric surface)

Given a ray in parametric form, i.e.

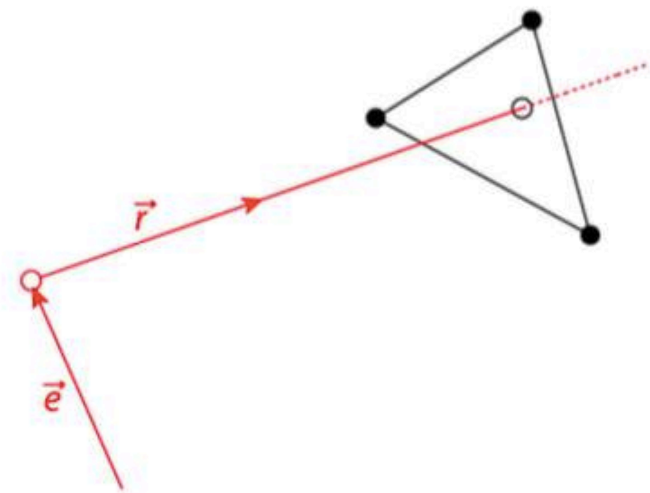
$$\vec{p}(t) = \vec{e} + t\vec{d}$$

and a surface in its parametric form, i.e.

$$f(u, v)$$

we can calculate the intersection point(s) by

$$\vec{e} + t\vec{d} = \vec{f}(u, v)$$



Ray-object intersection (parametric surface)

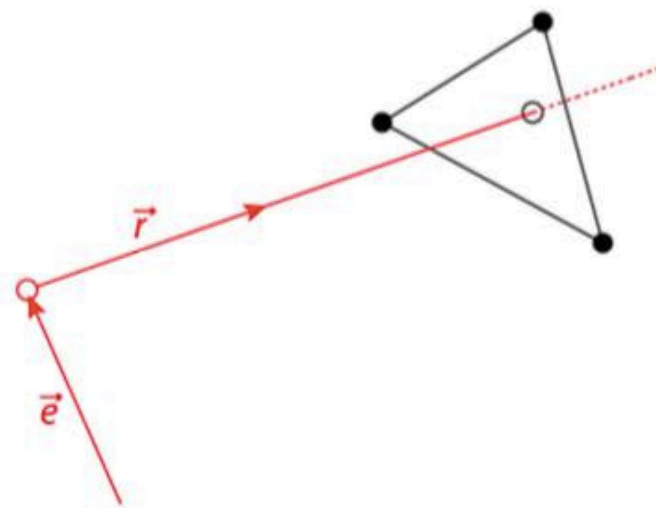
Notice that

$$\vec{e} + t\vec{d} = \vec{f}(u, v)$$

or

$$x_e + tx_d = f(u, v)$$
$$y_e + ty_d = f(u, v)$$
$$z_e + tz_d = f(u, v)$$

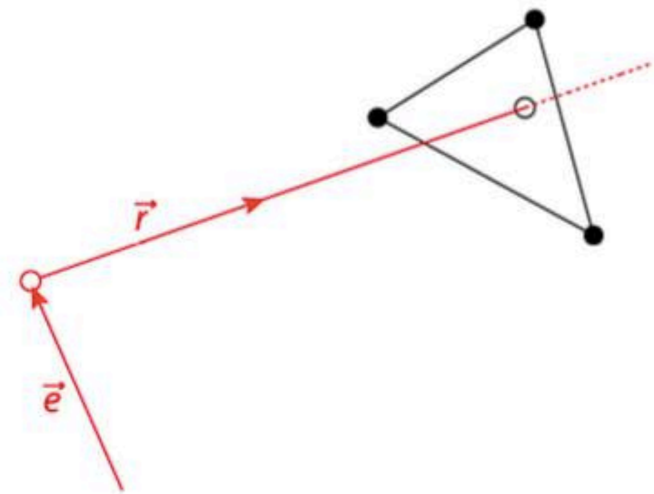
represents 3 equations with 3 unknowns (t, u, v), i.e. a linear equation system.



Ray-triangle intersection

This comes in very handy for ray-triangle intersections:

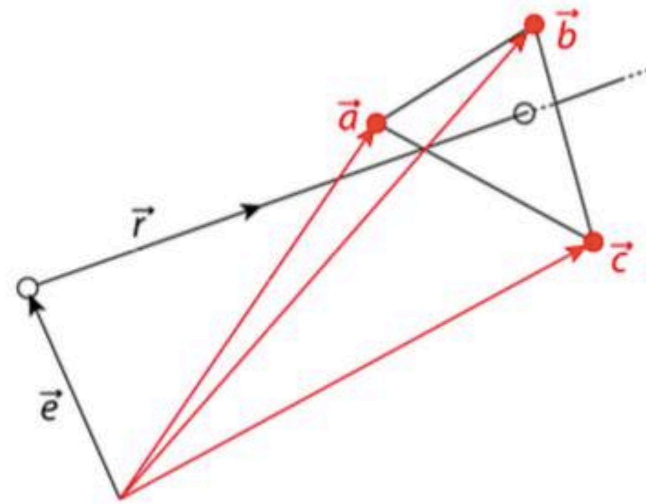
- We first calculate the intersection point of the ray with the plane defined by the triangle.
- Then we check if this point is within the triangle or not.



Plane specification

Recall that the plane V through the points $\vec{a}$, $\vec{b}$, and $\vec{c}$ can be written as

$$p(\vec{\beta}, \gamma) = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$



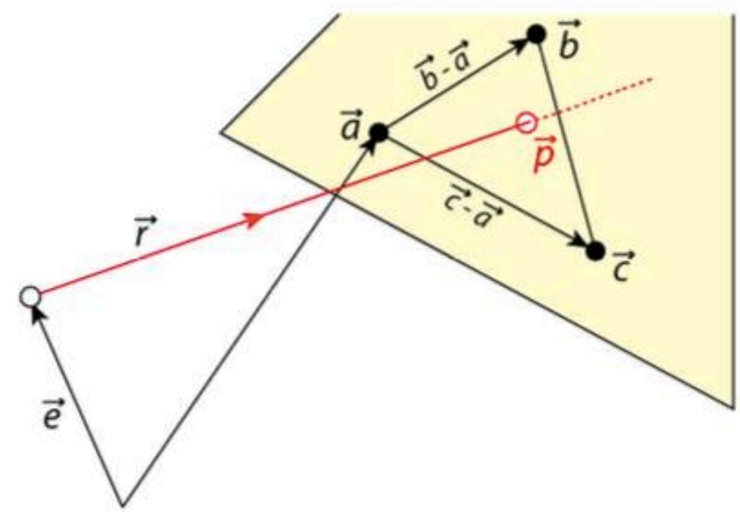
Ray-plane specification

Again, intersection points must fullfil the plane and the ray equation.

Hence, we get

$$\vec{e} + t\vec{d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

That give us ...



Ray-plane specification

...the following three equations

$$\begin{aligned} x_e + t x_d &= x_a + \beta(x_b - x_a) + \gamma(x_c - x_a) \\ y_e + t y_d &= y_a + \beta(y_b - y_a) + \gamma(y_c - y_a) \\ z_e + t z_d &= z_a + \beta(z_b - z_a) + \gamma(z_c - z_a) \end{aligned}$$

which can be rewritten as

$$\begin{aligned} (x_a - x_b)\beta + (x_a - x_c)\gamma + x_d t &= x_a - x_e \\ (y_a - y_b)\beta + (y_a - y_c)\gamma + y_d t &= y_a - y_e \\ (z_a - z_b)\beta + (z_a - z_c)\gamma + z_d t &= z_a - z_e \end{aligned}$$

or as

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

Ray-plane specification

If we write

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

as

$$A \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

then we see that

$$\begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = A^{-1} \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}$$

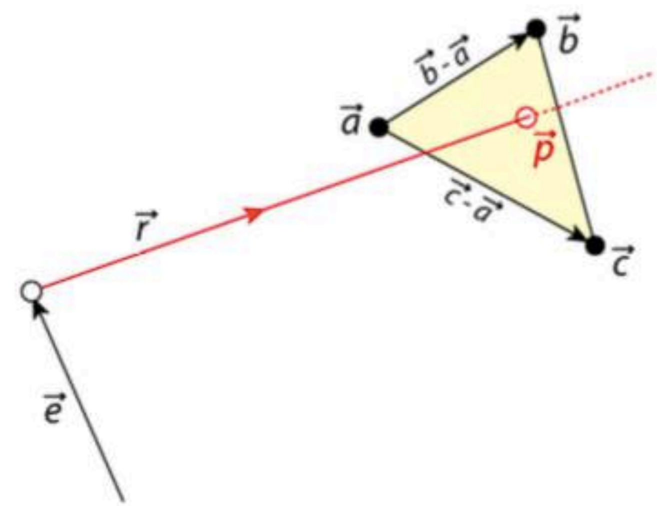
Rays: parametric representation

We can use t to calculate the intersection point $\vec{p}(t)$ (or β, γ to calculate $\vec{p}(\beta, \gamma)$).

But first, we can use β and γ to verify if it is inside of the triangle or not:

- $\beta > 0$
- $\gamma > 0$
- $\beta + \gamma < 1$

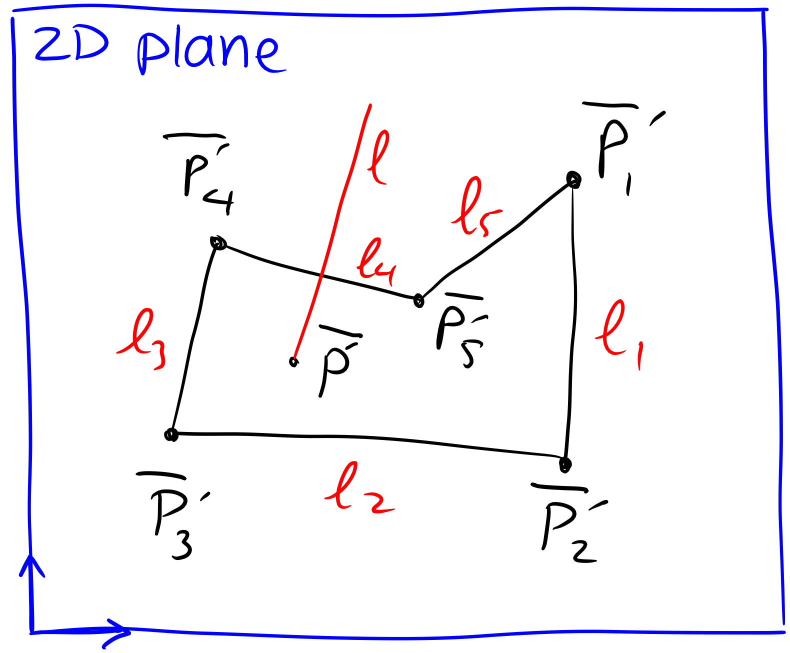
because we can interpret these as barycentric coordinates.



Computing Ray-Poly Intersections: Step b

Key theorem:

If $\bar{p}'$ inside, every 2D half-line starting at $\bar{p}'$ must intersect the polygon's boundary an <u>odd</u> # of times



Verification algorithm:

① pick any non-vertex point on boundary (e.g. $\bar{q}' = \frac{1}{2}(\bar{p}'_1 + \bar{p}'_2)$)

② define lines ℓ through $\bar{p}', \bar{q}'$ and ℓ_i through $\bar{p}'_i, \bar{p}'_{i+1}$

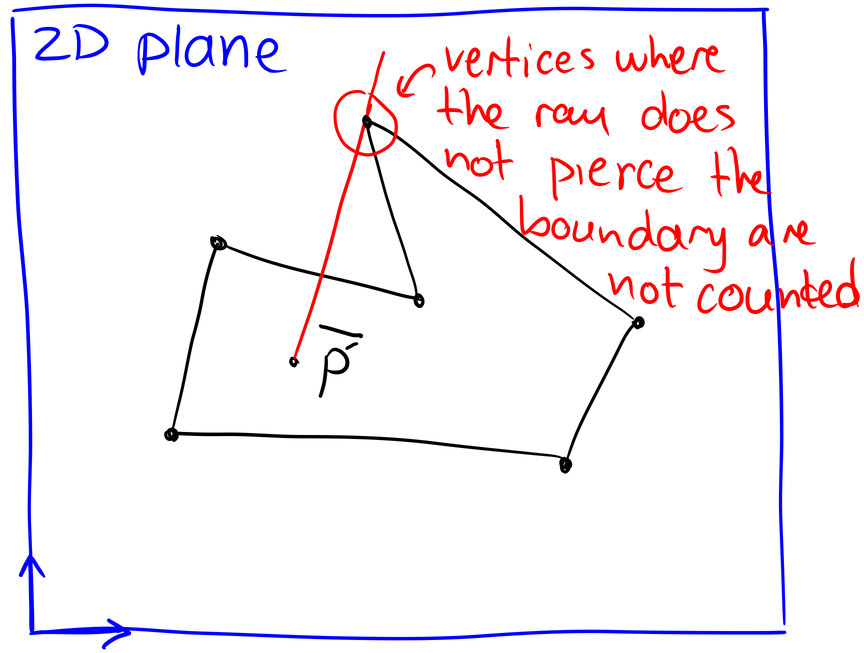
③ intersect ℓ with each ℓ_i

④ count intersections that are
   ⓐ on same side of $\bar{p}'$, and
   ⓑ on polygon boundary

Computing Ray-Poly Intersections: Step b

Key theorem:

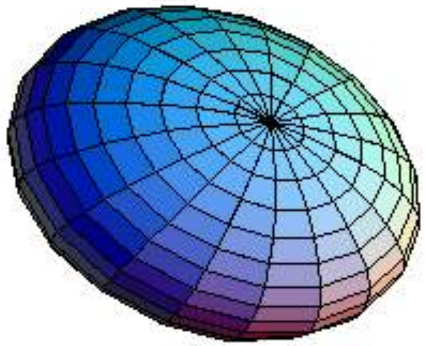
If $\bar{p}'$ inside, every 2D half-line starting at $\bar{p}'$ must intersect the polygon's boundary an <u>odd</u> # of times

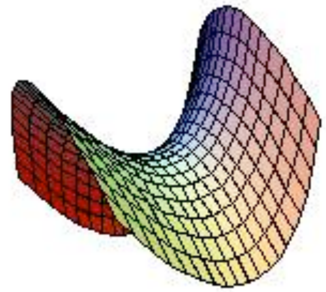


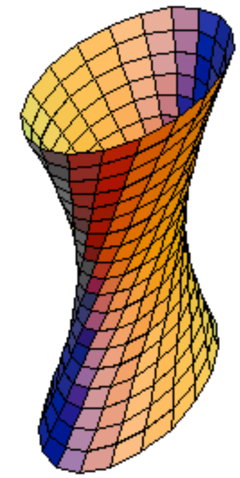
Verification algorithm:

1. pick any non-vertex point on boundary (eg. $\bar{q}' = \frac{1}{2}(\bar{P}_1' + \bar{P}_2')$)
2. define lines ℓ through $\bar{p}', \bar{q}'$ and ℓ_i through $\bar{P}_i', \bar{P}_{i+1}'$
3. intersect ℓ with each ℓ_i
4. count intersections ← counting is a bit more involved if line ℓ intersects a polygon vertex

Computing Ray-Quadric Intersections



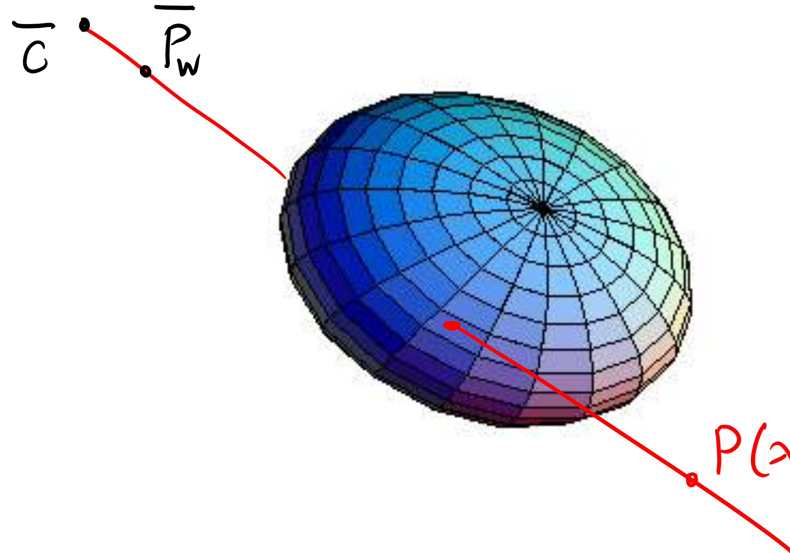




General implicit equation

$$[x\ y\ z\ 1]\underbrace{\begin{bmatrix} A & D & E & G \\ D & B & F & H \\ E & F & C & I \\ G & H & I & J \end{bmatrix}}_{\text{defined up to a scale factor}}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

Computing Ray-Quadric Intersections



$$P(\lambda) = \bar{c} + \lambda(\bar{P}_w - \bar{c})$$

Must solve the equation

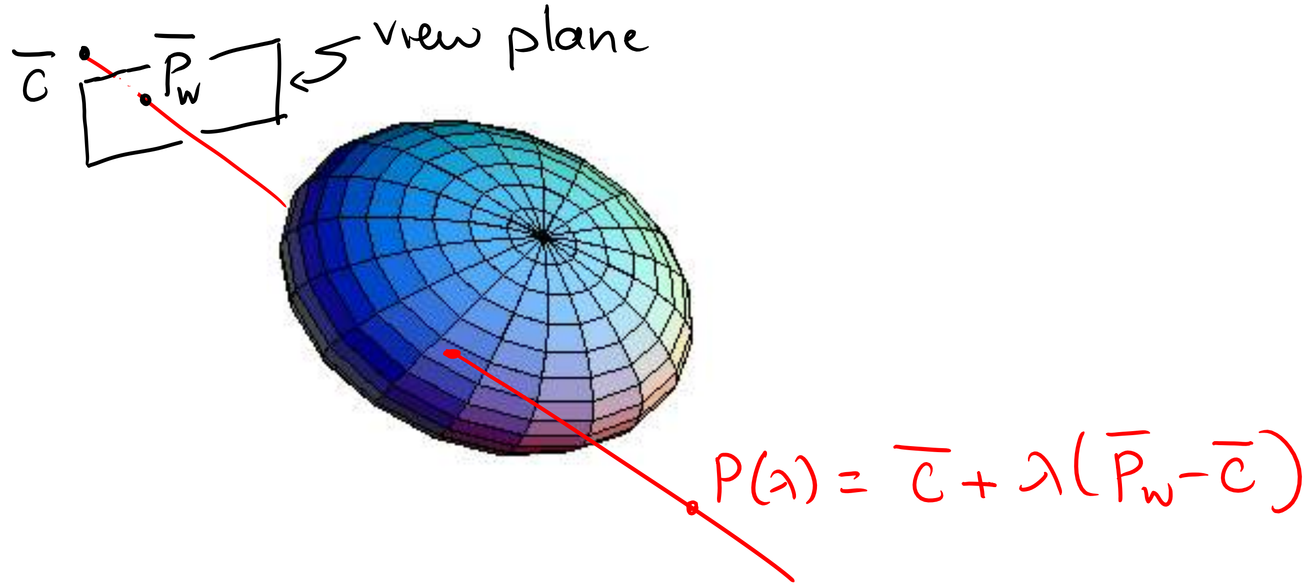
$$P(\lambda)^T \begin{bmatrix} A & D & E & G \\ D & B & F & H \\ E & F & C & I \\ G & H & I & J \end{bmatrix} P(\lambda) = 0$$

the only unknown is λ

expressed in homogeneous 3D coords

for a given quadric, this matrix is known

Computing Ray-Quadric Intersections: 3 Cases



$\Delta > 0$
$\Rightarrow$ 2 "hits"

$\Delta < 0$
$\Rightarrow$ 0 "hits"

$\Delta = 0$
$\Rightarrow$ 1 "hit"

after expanding, we have a quadratic equation in terms of λ:

$$\alpha\lambda^2 + \beta\lambda + \gamma = 0$$

solution is $\lambda = \dfrac{-\beta \pm \sqrt{\Delta}}{2\alpha}$, $\Delta = \beta^2 - 4\alpha\gamma$

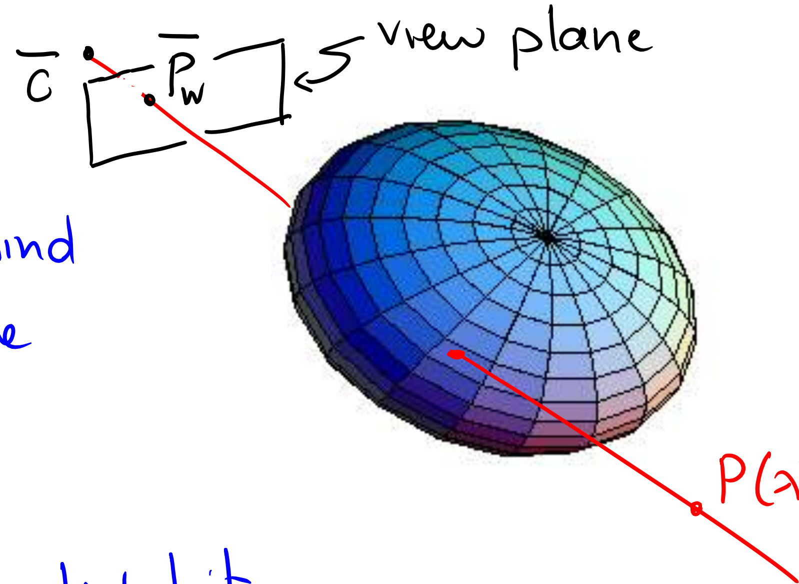
Ray-Quadric Intersections: Sub-cases for Δ>0

$\lambda_1, \lambda_2 < 0$

$\Rightarrow$ hits are behind the view plane

$\lambda_1 > 0,\ \lambda_2 > 0$

$\Rightarrow$ 2 valid hits, smallest λ gives intersection closest to camera



$P(\lambda) = \bar{c} + \lambda(\bar{p}_w - \bar{c})$

$\lambda_1 > 0,\ \lambda_2 < 0$

$\Rightarrow P(\lambda_1)$ is a valid hit

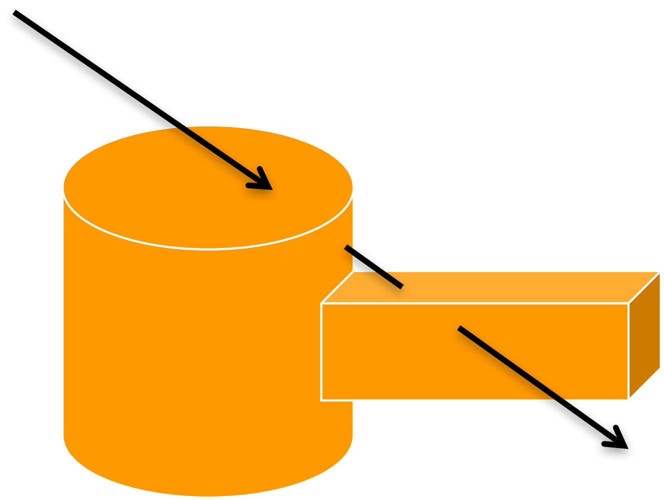
after expanding, we have a quadratic equation in terms of λ:

$$\alpha \lambda^2 + \beta \lambda + \gamma = 0$$

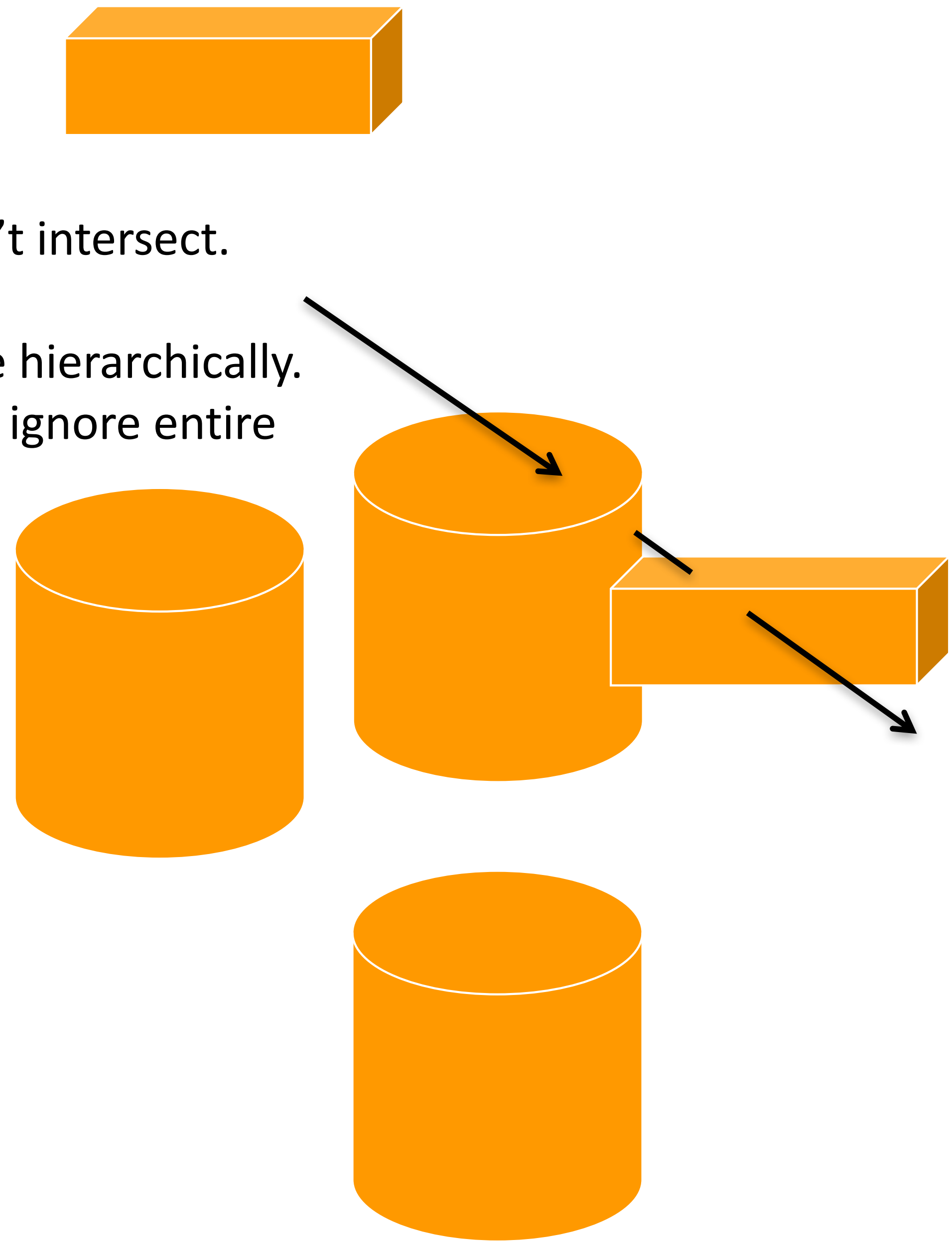
solution is $\lambda = \dfrac{-\beta \pm \sqrt{\Delta}}{2\alpha}$, $\Delta = \beta^2 - 4\alpha\gamma$

Intersecting Rays & Composite Objects

- Intersect ray with component objects
- Process the intersections ordered by depth to return intersection pairs with the object.



Ray Intersection: Efficiency Considerations



Speed-up the intersection process.

- Ignore object that clearly don't intersect.
- Use proxy geometry.
- Subdivide and structure space hierarchically.
- Project volume onto image to ignore entire

Sets of rays.