Implicit Curve and Surfaces

Prof. Fabio Pellacini
[some slide ideas by Prof. Wojciech Jarosz]

© 2020 Fabio Pellacini •

Implicit curves and surfaces

- Parametric models, meshes and subdivs are shape representations that explicitly represent the curve points and surface boundaries
- Implicit curves and surfaces are an alternative shape representation where boundaries are implicitly defined as the isovalues of a function

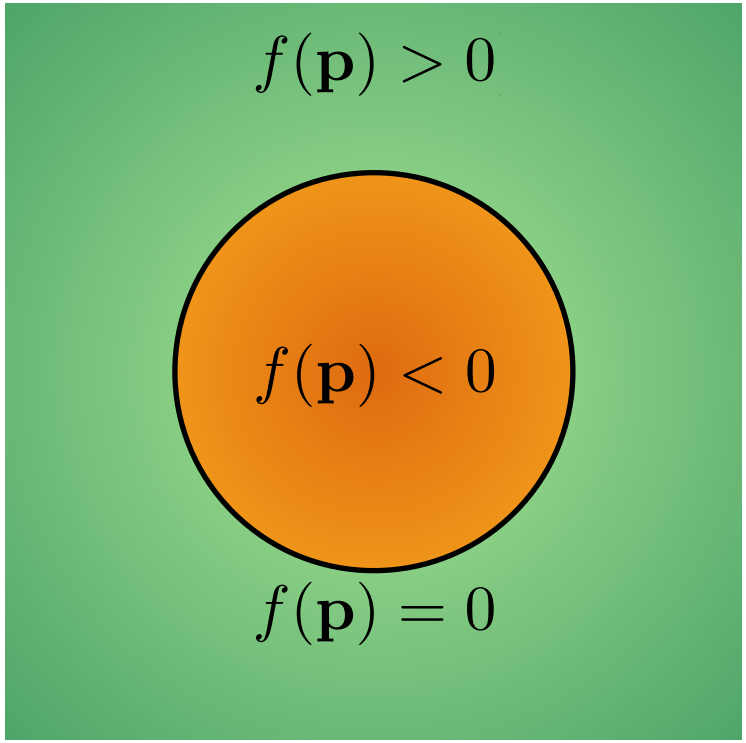
$$S = \{\mathbf{p} | f(\mathbf{p}) = 0\}$$

© 2020 Fabio Pellacini •

Signed Distance Functions

- The most used implicit representation is to consider functions that encode the signed distance to the boundary
  - The sign indicates whether we are outside or inside the surface

$$O = \{\mathbf{p} | f(\mathbf{p}) > 0\} \quad \text{outside}$$
$$S = \{\mathbf{p} | f(\mathbf{p}) = 0\} \quad \text{curve/surface}$$
$$I = \{\mathbf{p} | f(\mathbf{p}) < 0\} \quad \text{indice}$$

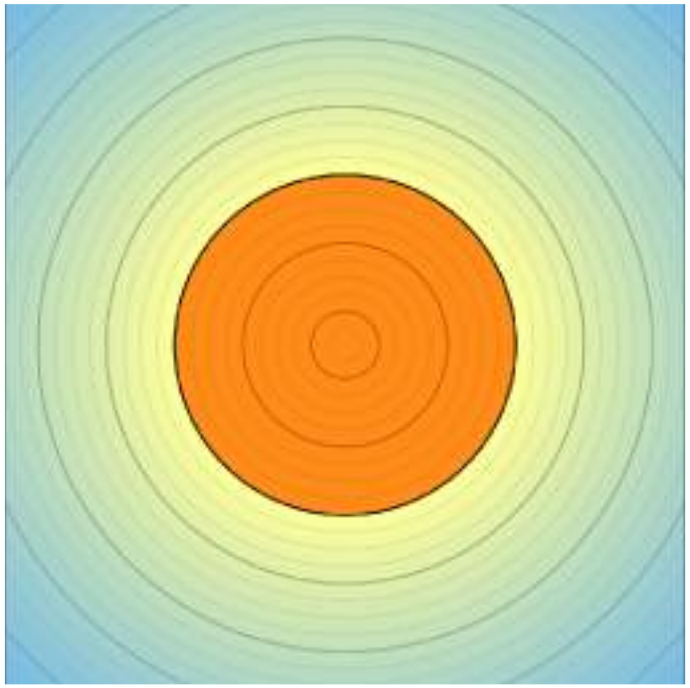


© 2020 Fabio Pellacini

Signed Distance Functions

- Example for circles and spheres

$$f(\mathbf{p}) = |\mathbf{p} - \mathbf{c}| - r$$



© 2020 Fabio Pellacini

Signed Distance Functions

- Normals of SDFs are computed as the gradient of the function

$$\mathbf{n} = \nabla f(\mathbf{p}) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) (\mathbf{p})$$

- Since the gradient is often not available analytically, we estimate it numerically by finite differences

$$n_x \approx \frac{f(p_x + h) - f(p_x - h)}{2h}$$

$$n_y \approx \frac{f(p_y + h) - f(p_y - h)}{2h}$$

$$n_z \approx \frac{f(p_z + h) - f(p_z - h)}{2h}$$

© 2020 Fabio Pellacini

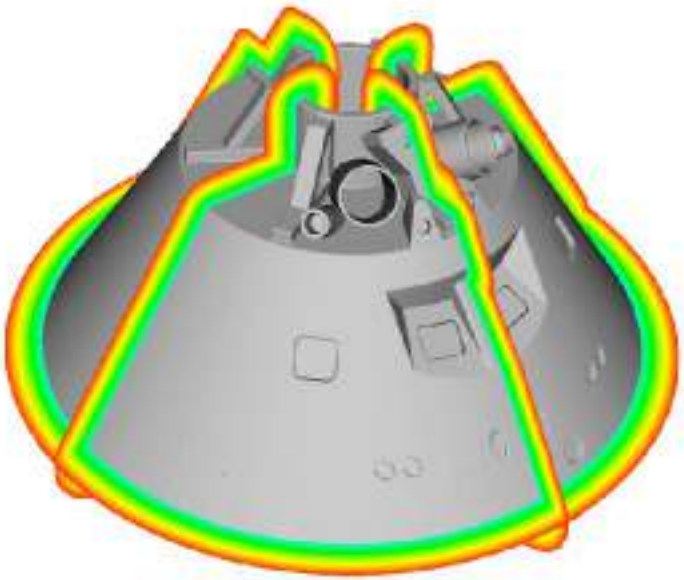
Signed Distance Functions

- As for many numerical approximation we need to pay attention to numerical precision, in this case for the choice of increment when computing normals

© 2020 Fabio Pellacini

Signed Distance Functions

- SDFs for surfaces are represented in all points in a volume
- To keep memory usage limited, real implementations store SDFs only in a narrow band around the surface using sparse hash grids



[https://al-ro.github.io/projects/sdf/]

© 2020 Fabio Pellacini •

Signed Distance Functions

- Pros: easy and robust determination of inside/outside
  - used in collision detection for animation
- Pros: easy to determine if a point is on the surface
  - used in raytracing
- Cons: hard to enumerate points on the surfaces
- Cons: hard to modify directly

- Applications: Great for procedural modeling
- Applications: Very useful in deep learning

© 2020 Fabio Pellacini

Implicit Modeling

© 2020 Fabio Pellacini

[Inigo Quilez – https://www.shadertoy.com/view/4tByz3]

© 2020 Fabio Pellacini

[Inigo Quilez – https://www.shadertoy.com/view/ldScDh]

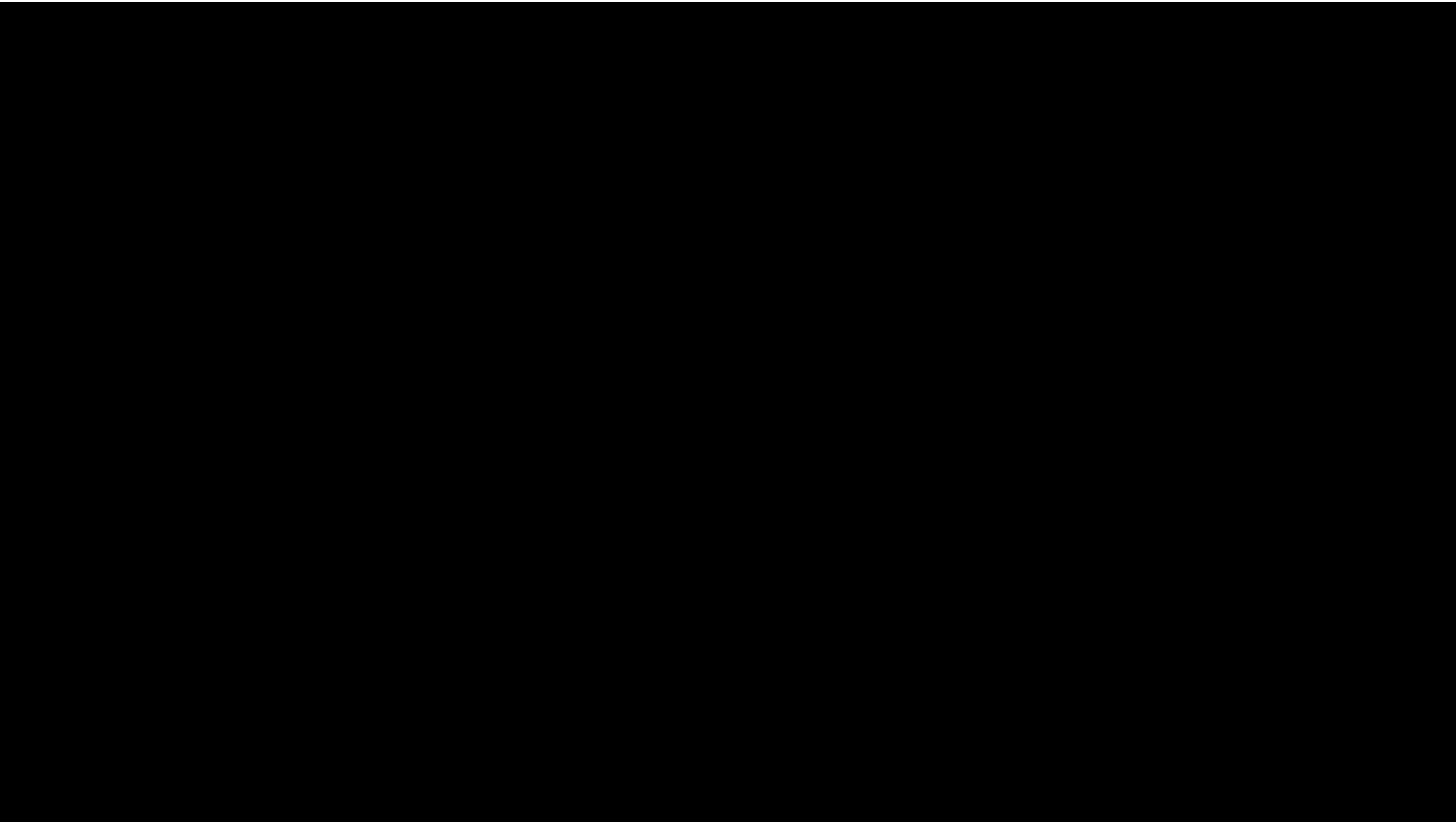
© 2020 Fabio Pellacini

[Inigo Quilez – https://www.shadertoy.com/view/4ttSWf]

© 2020 Fabio Pellacini

[Inigo Quilez – https://www.shadertoy.com/view/3lsSzf]

© 2020 Fabio Pellacini



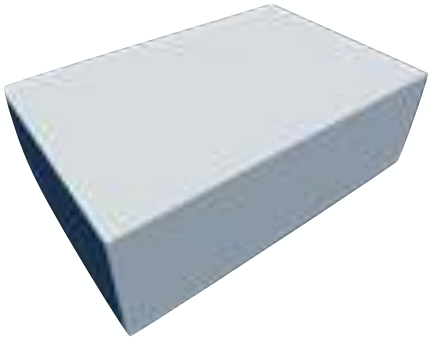
[Inigo Quilez – https://www.shadertoy.com/view/3lsSzf]

© 2020 Fabio Pellacini

Primitive Shapes

- Create objects by combining basic shapes and sculpting
- Primitive shapes are all shapes you can analytically compute distance
- A good list can be found at http://www.iquilezles.org

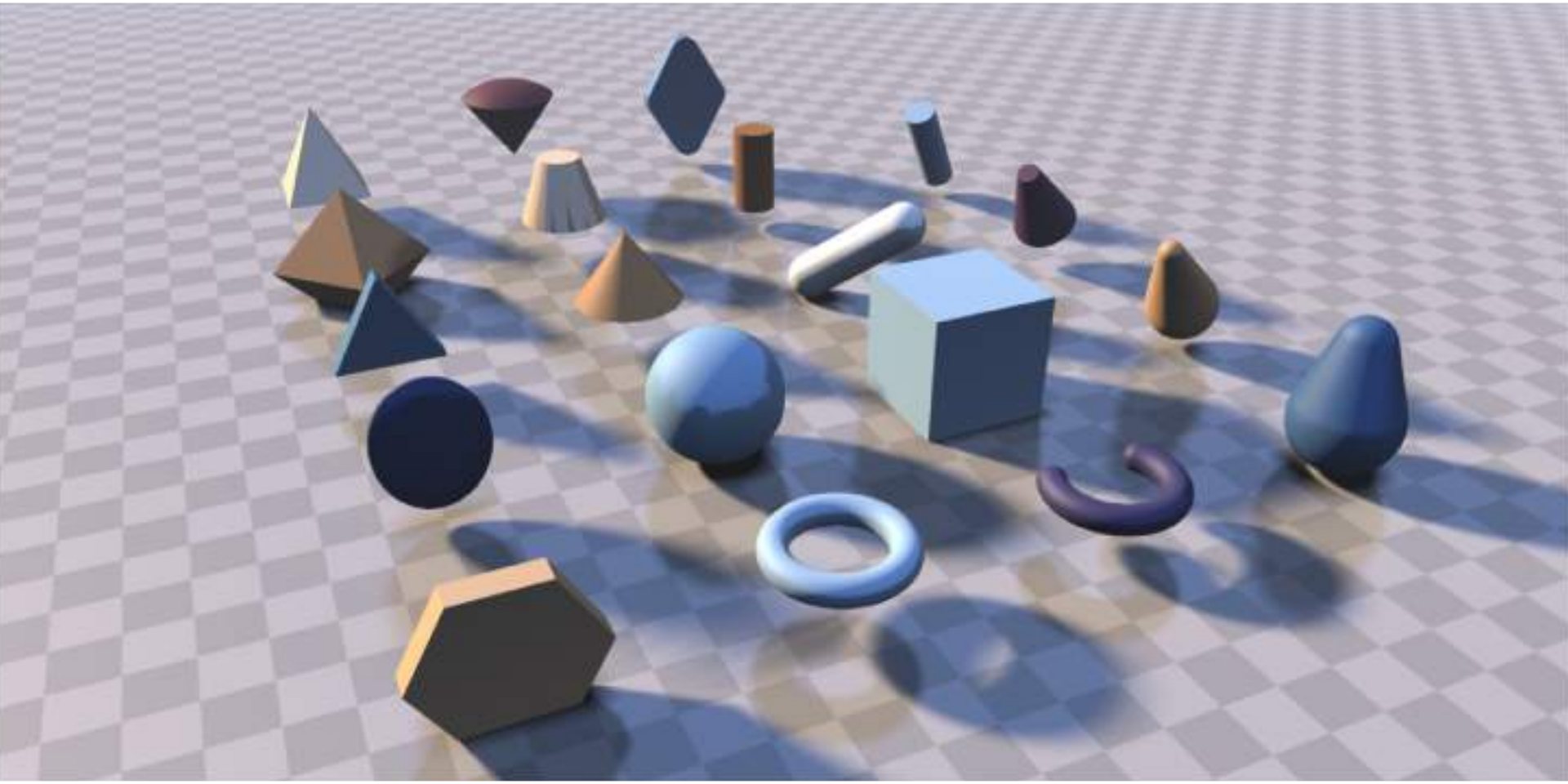
```
float sdSphere(vec3f p, float s) { return length(p)-s; }
float sdBox(vec3f p, vec3f b) { vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0); }
float sdCylinder(vec3f p, float h, float r) {
  vec2 d = abs(vec2(length(p.xz),p.y)) - vec2(h,r);
  return min(max(d.x,d.y),0.0) + length(max(d,0.0)); }
```



© 2020 Fabio Pellacini

Primitive Shapes

- Primitive shapes are all shapes you can analytically compute distance

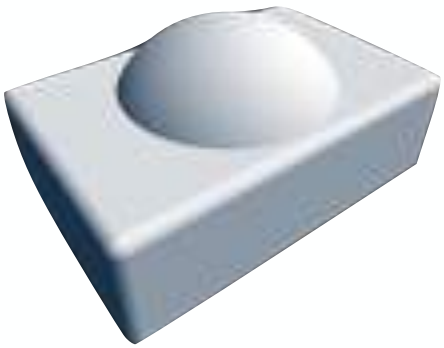


© 2020 Fabio Pellacini

Boolean Operations

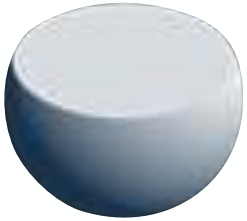
- Boolean operations in modeling are defined over object volumes
- We are interested in computing the surface of the resulting volume
- With SDFs we can trivially compute booleans

Union



Subtraction

Intersection

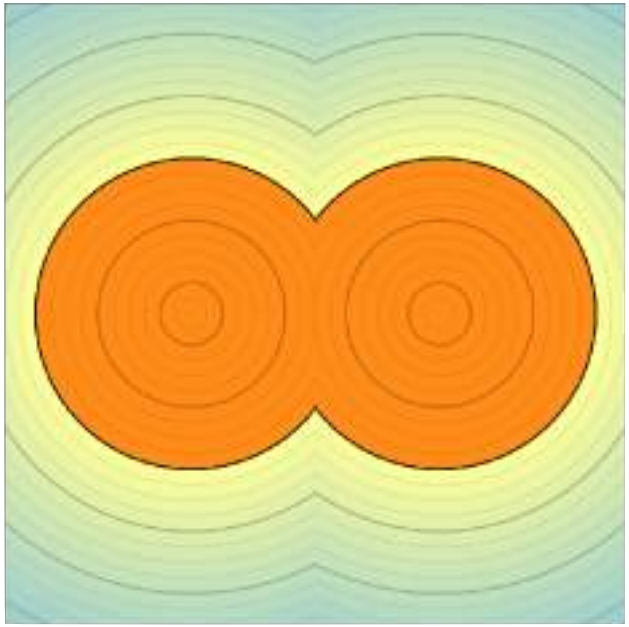


© 2020 Fabio Pellacini

Boolean Operations

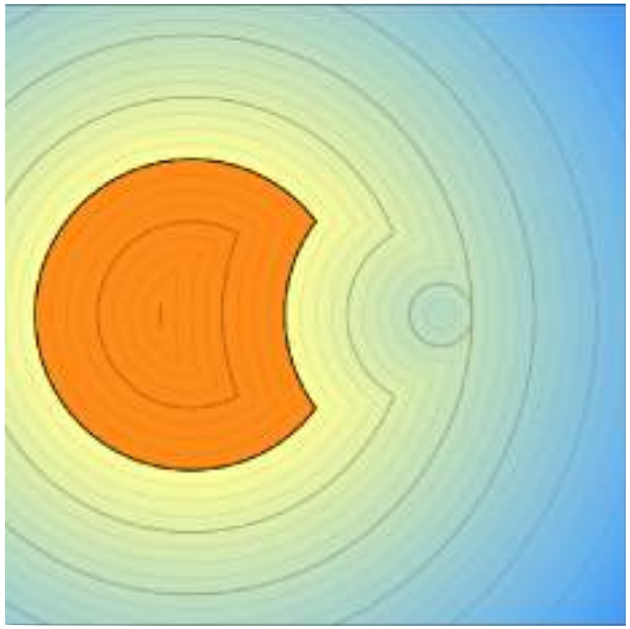
- Boolean can be defined by simply combining SDFs with min/max operations

Union



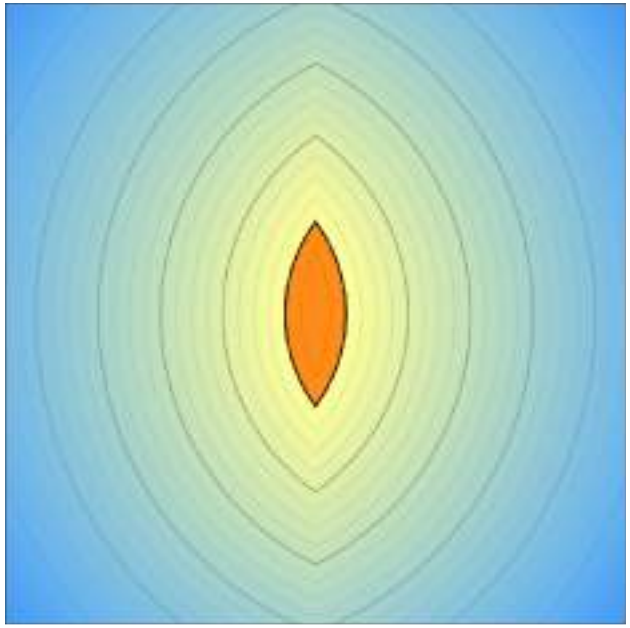
$$f_u(\mathbf{p}) = \min(f_1(\mathbf{p}), f_2(\mathbf{p}))$$

Subtraction



$$f_s(\mathbf{p}) = \max(f_1(\mathbf{p}), -f_2(\mathbf{p}))$$

Intersection



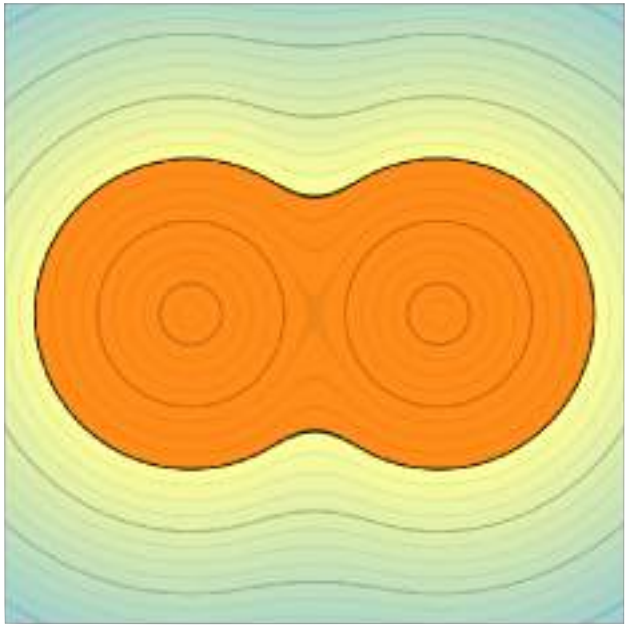
$$f_i(\mathbf{p}) = \max(f_1(\mathbf{p}), f_2(\mathbf{p}))$$

© 2020 Fabio Pellacini

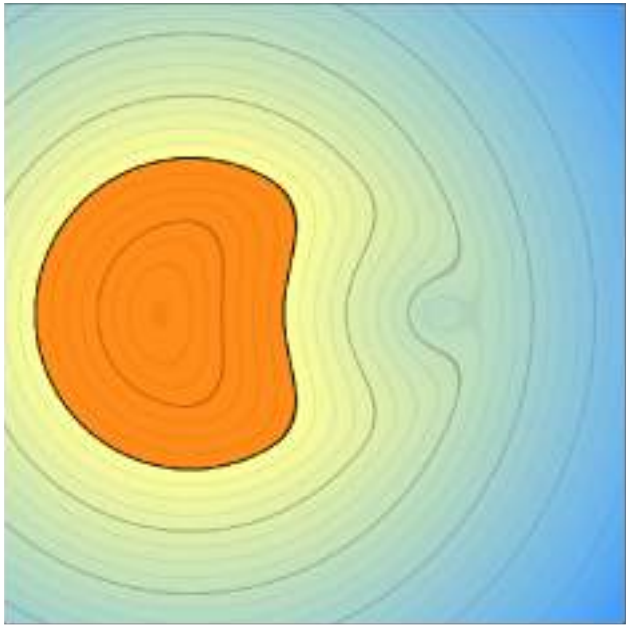
Smooth Boolean Operations

- Boolean operations can be generalized to perform smooth blending between shapes by introducing blending functions

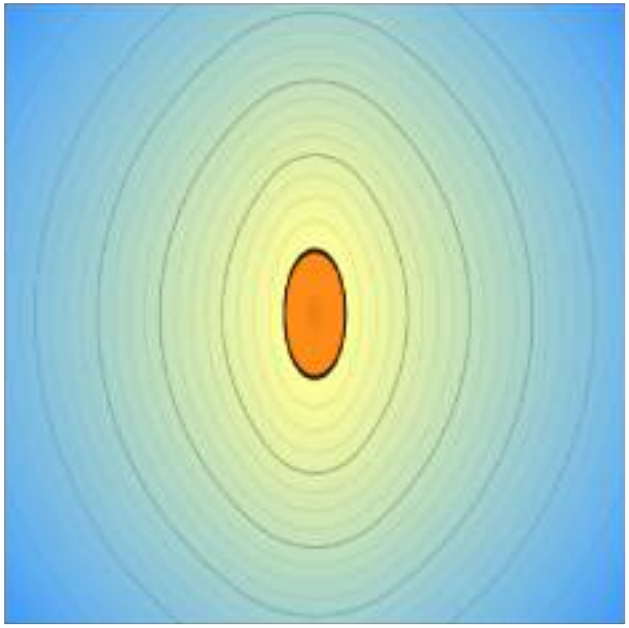
"Smooth" Union



"Smooth" Subtraction



"Smooth" Intersection

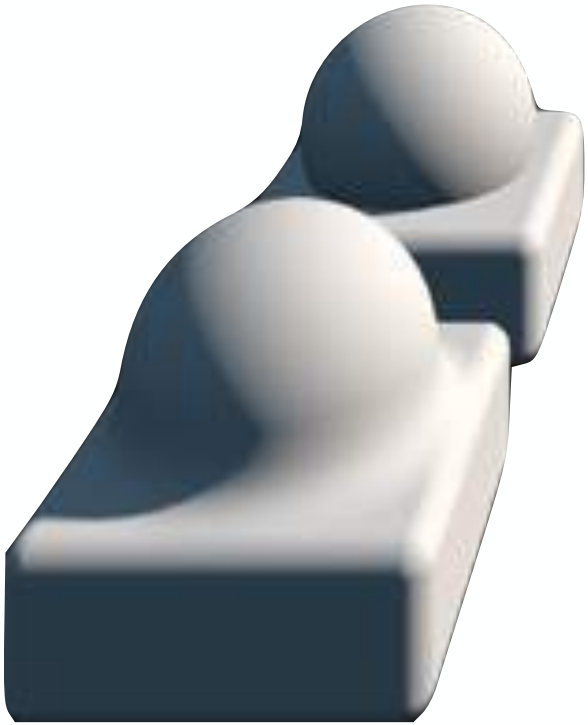


© 2020 Fabio Pellacini

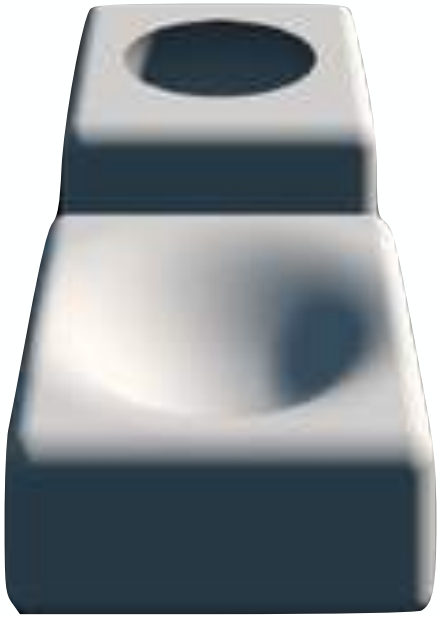
Smooth Boolean Operations

- Boolean operations can be generalized to perform smooth blending between shapes by introducing blending functions

Union



Subtraction



Intersection

© 2020 Fabio Pellacini

Smooth Boolean Operations

- Boolean operations can be generalized to perform smooth blending between shapes by introducing blending functions

© 2020 Fabio Pellacini

Boolean Operations

```
float opUnion( float d1, float d2 ) {  min(d1,d2); }

float opSubtraction( float d1, float d2 ) {
  return max(-d1,d2); }

float opIntersection( float d1, float d2 ) {
  return max(d1,d2); }

float opSmoothUnion( float d1, float d2, float k ) {
  float h = clamp( 0.5 + 0.5*(d2-d1)/k, 0.0, 1.0 );
  return mix( d2, d1, h ) - k*h*(1.0-h); }

float opSmoothSubtraction( float d1, float d2, float k ) {
  float h = clamp( 0.5 - 0.5*(d2+d1)/k, 0.0, 1.0 );
  return mix( d2, -d1, h ) + k*h*(1.0-h); }

float opSmoothIntersection( float d1, float d2, float k ) {
  float h = clamp( 0.5 - 0.5*(d2-d1)/k, 0.0, 1.0 );
  return mix( d2, d1, h ) + k*h*(1.0-h); }
```


© 2020 Fabio Pellacini

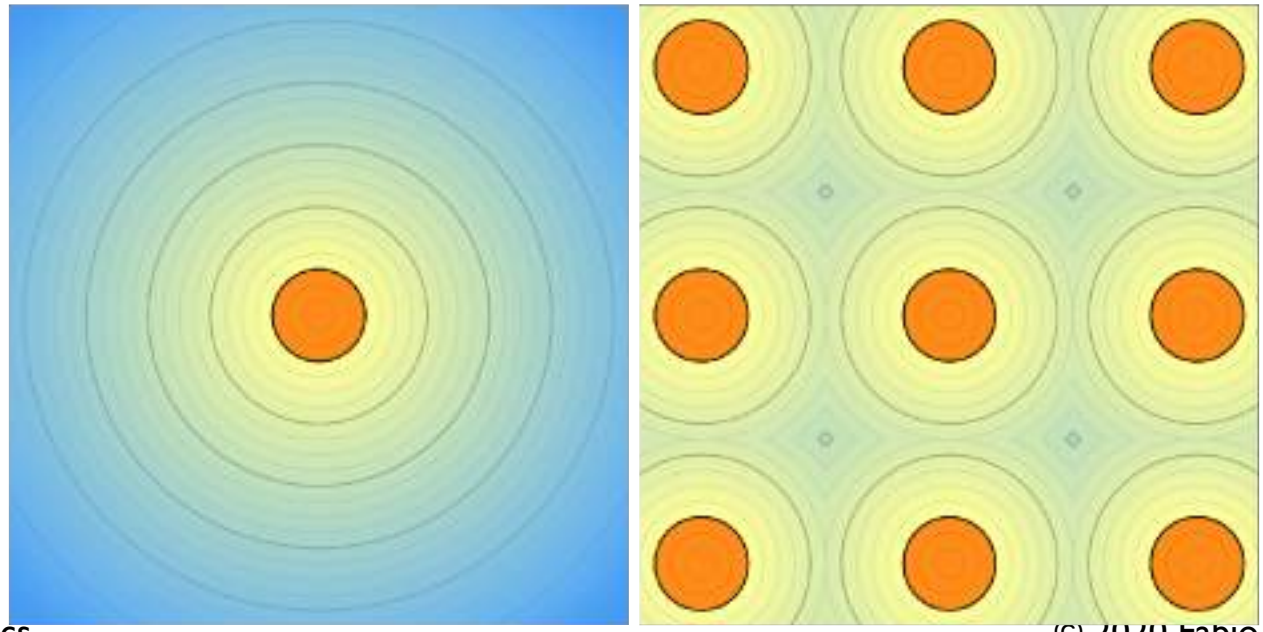
Transforms and Repetition

- We can transform an SDF by querying the function with the point inversely transformed

$$Tf(\mathbf{p}) = f(T^{-}1\mathbf{p})$$

- We can repeat a shape by taking the modulus of the point

$$f_r(\mathbf{p}, \mathbf{c}) = f(\mathrm{mod}(\mathbf{p} + \mathbf{c}/2, \mathbf{c}) - \mathbf{c}/2)$$



© 2020 Fabio Pellacini

Rendering Implicit Surfaces

© 2020 Fabio Pellacini

Rendering Implicit Surfaces

- Method 1: convert SDF to poly lines (2D) or meshes (3D)
  - Many well-studied algorithms to do this
  - Different tradeoffs
  - All quite tricky to implement correctly, so won't cover in details
- Method 2: retrace SDFs directly
  - Fairly robust method
  - Still not we'll studied for all cases

© 2020 Fabio Pellacini

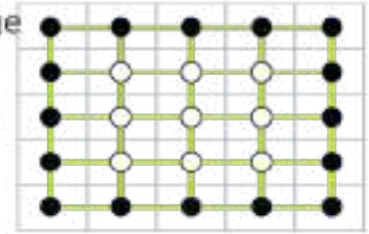
Marching squares

1	1	1	1	1
1	2	3	2	1
1	3	3	3	1
1	2	3	2	1
1	1	1	1	1

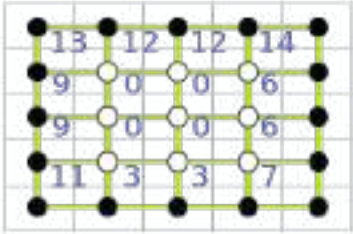
Threshold with iso-value

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

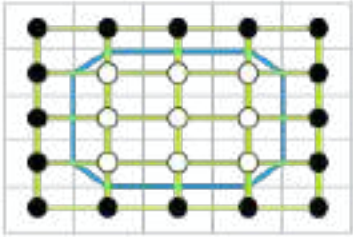
Binary image to cells



Give every cell a number based on which corners are true/false



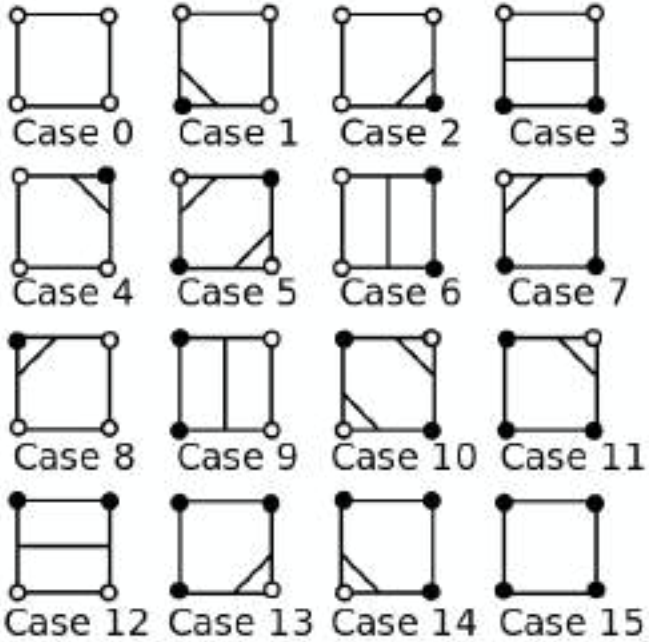
Look up the contour lines in the database and put them in the cells



Look at the original values and use linear interpolation to determine a more accurate position of all the line end-points

1	1	1	1	1
1	2	3	2	1
1	3	3	3	1
1	2	3	2	1
1	1	1	1	1

Look-up table contour lines



[Wikipedia]

© 2020 Fabio Pellacini

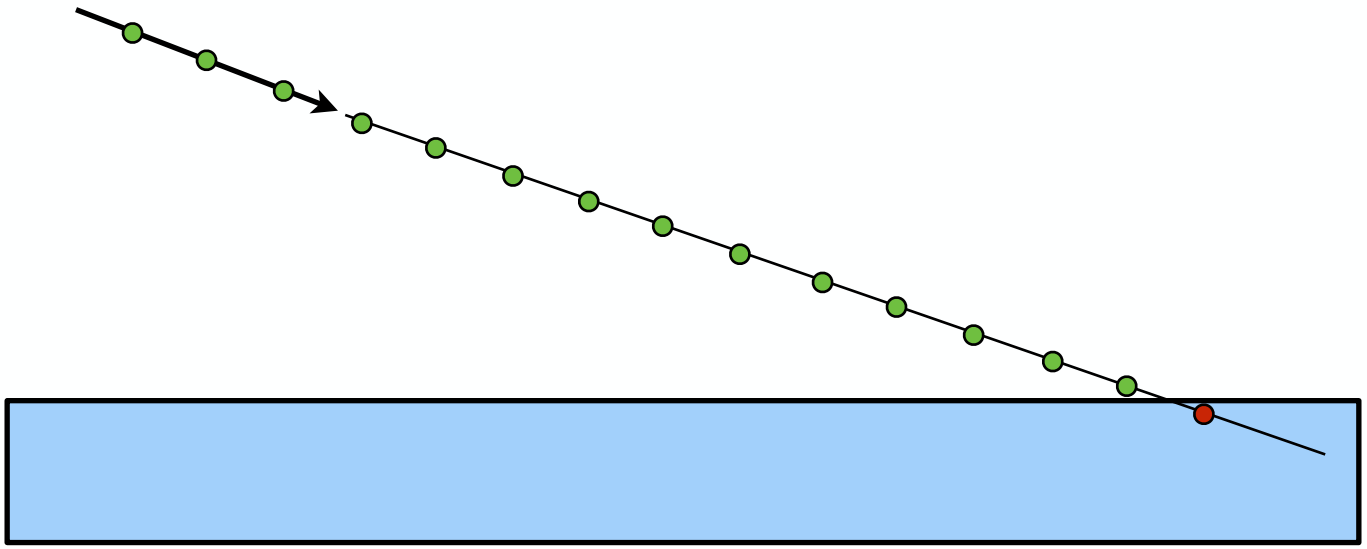
Ray tracing implicits

- For models that are boolean combination of analytics primitives, we could compute the solution numerically since method exists for all those surfaces
- But that would be restrictive and would not work for general SDF
- The most common solution is to solve the problem numerically
  - Several methods exist in practice

© 2020 Fabio Pellacini

Ray marching

- The simplest method is to walk along the ray at fixed increments and stop when the SDF is negative
- This method works in all cases, but it is not very accurate nor very fast



© 2020 Fabio Pellacini

Ray marching

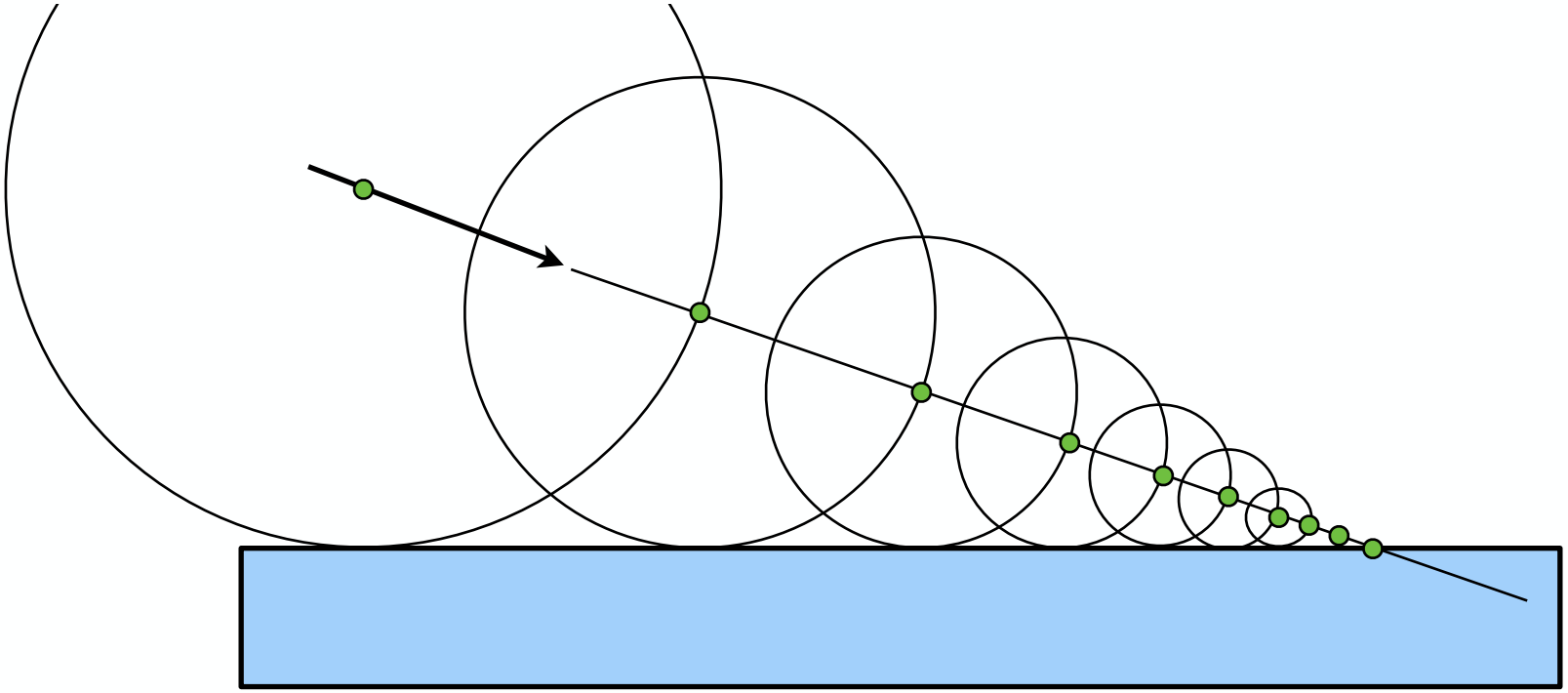
- The simplest method is to walk along the ray at fixed increments and stop when the SDF is negative
- This method works in all cases, but it is not very accurate nor very fast

```
bool raytrace(ray3f ray, SDF* sdf, float* dist) {
  auto t = ray.tmin;
  while(t < ray.tmax) {
    auto p = eval_ray(ray, t);
    if(eval_sdf(sdf, p) < 0) { *dist = t; return true; }
    t += step_size;
  }
  return false;
}
```


© 2020 Fabio Pellacini

Sphere tracing

- The key to make ray marching faster and more accurate is to step along the ray each time with the longest distance we know it is safe
- That distance is just the SDF itself evaluated at the point we are at



© 2020 Fabio Pellacini

Sphere tracing

- The key to make ray marching faster and more accurate is to step along the ray each time with the longest distance we know it is safe
- That distance is just the SDF itself evaluated at the point we are at

```
bool spheretrace(ray3f ray, SDF* sdf, float* dist) {
  auto t = ray.tmin;
  while(t < ray.tmax) {
    auto p = eval_ray(ray, t);
    auto d = eval_sdf(sdf, p);
    if(d < epsilon) { *dist = t; return true; }
    t += d;
  }
  return false;
}
```


© 2020 Fabio Pellacini

Texture mapping

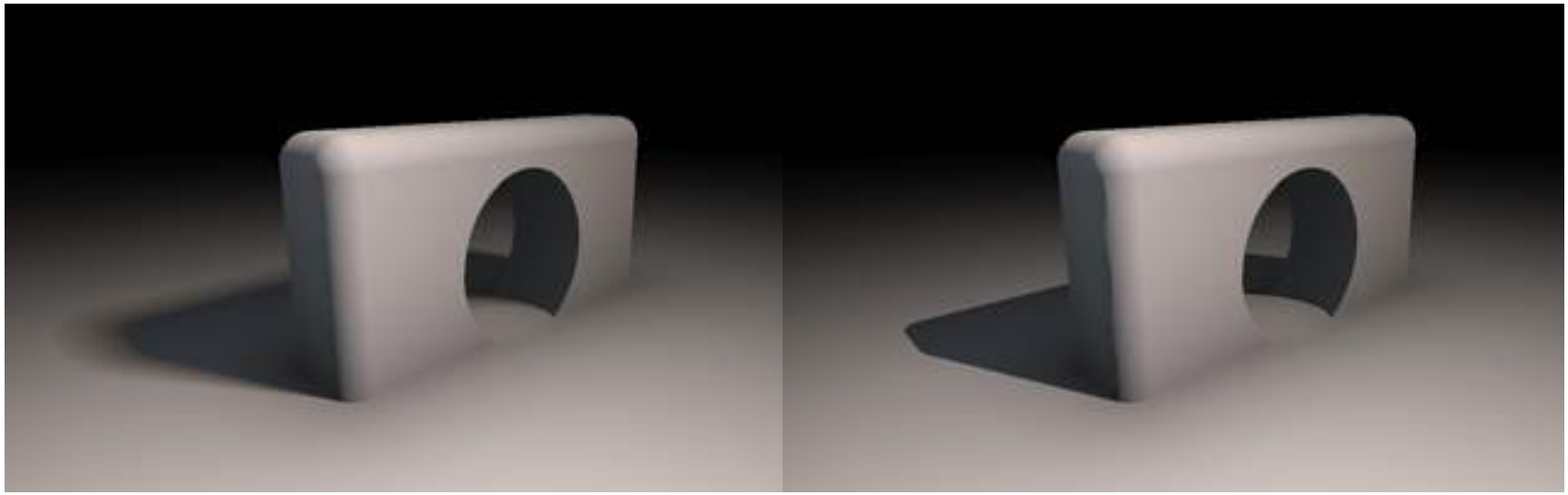
- SDFs do not define texture coordinates and support booleans
- Instead of applying 2D texture, we apply 3D textures to them
- 3D texture are defined in the volume so we can look them up at any position
- Just like real-world objects that have the inside filled in

© 2020 Fabio Pellacini

[http://lernertandsander.com/cubes]

Lighting SDFs

- Since we can retraced them, we can use any lighting algorithm
- We can also cheat lighting by taking advantage of the SDF nature



© 2020 Fabio Pellacini

Dreams

© 2020 Fabio Pellacini

© 2020 Fabio Pellacini

© 2020 Fabio Pellacini

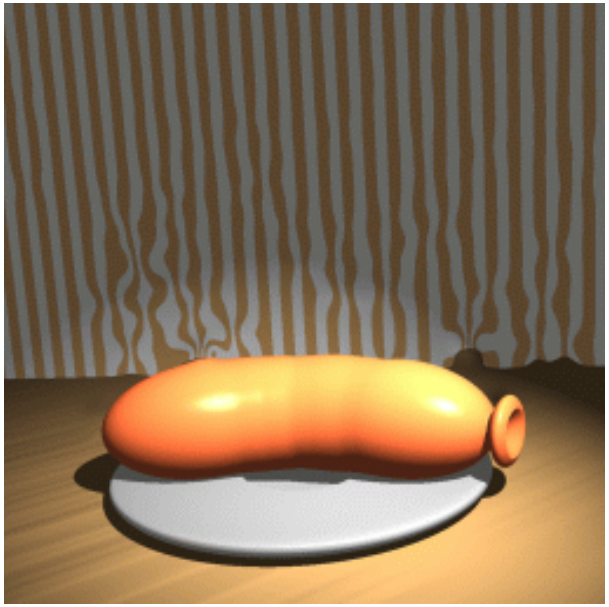
Path Tracing

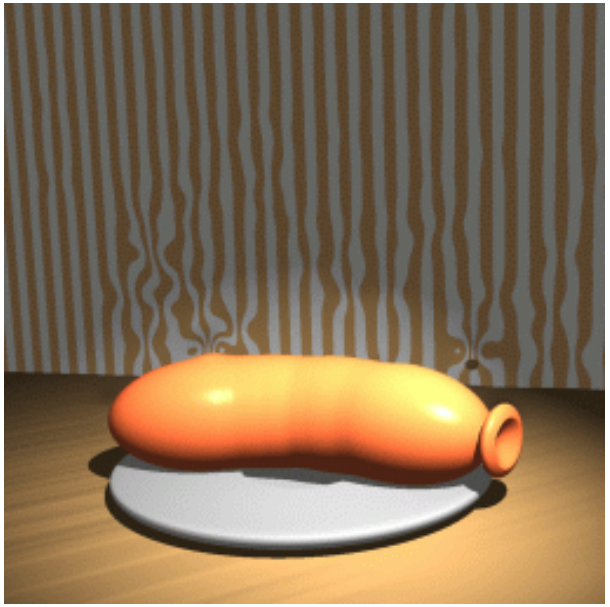
Prof. Fabio Pellacini

© 2020 Fabio Pellacini •

Models of light

- Geometric optics
  - light travel in straight lines
  - light particles, i.e. photons, do not interact with each other
  - describes: emission, reflection/refraction, absorption



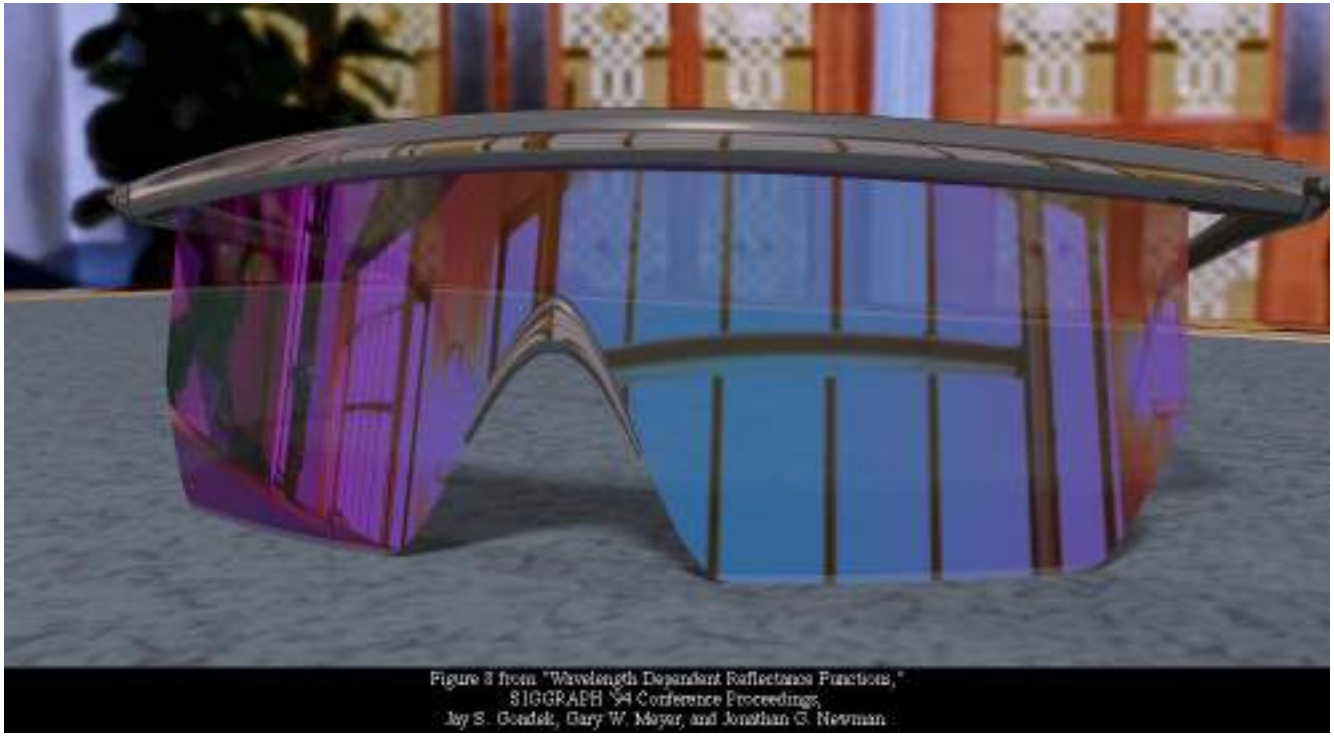


[Stam et al., 1996]

© 2020 Fabio Pellacini

Models of light

- Wave optics
  - light particles interact with each other
  - describes: diffraction, interference, polarization

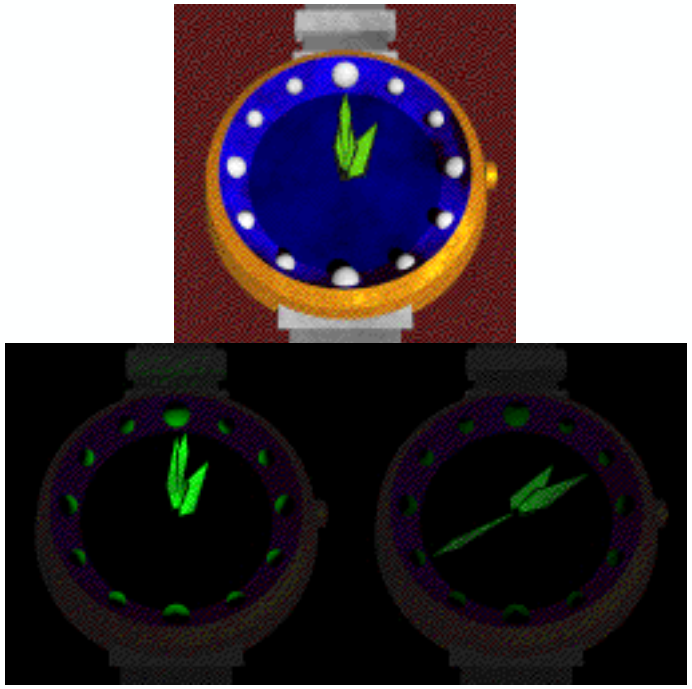


[Gondek et al., 1997]

© 2020 Fabio Pellacini

Models of light

- Quantum optics
  - light particles are like any other quantum particle
  - describes: fluorescence, phosphorescence



[Glassener et al., 1997]

© 2020 Fabio Pellacini

Real-world image formation

- Assumption: geometric optics
- Images are formed by a continuous process
  - photons are emitted from light sources
  - they travel in straight lines until the hit a surface
  - at the surfaces, they are either absorbed or reflected
  - the reflected ones continue the same process
  - eventually, some hit the eye
  - the eye measures how many photons hit it

© 2020 Fabio Pellacini

Physically-based rendering

- *All realistic image synthesis methods* simulate, to a different degree of approximation, the physics of lights
  - main idea: simulating the physics of light ensures realistic images
    - obviously we also need realistic input scenes
  - some algorithms simulate "virtual photons", tracing them from the lights to the eye as real photons would behave

- *Rendering equation:* describes formally the physics of light
  - common formulation based on geometric optics
  - describes nearly all illumination effects and surface materials

© 2020 Fabio Pellacini

Physically-based rendering

- *Path tracing:* an algorithm that solves the rendering equation
  - produces *physically-correct* images
  - simulates the physics of light using backwards raytracing, i.e. tracing paths from eye to lights
  - used in architecture, product design and movies
  - drawback: slow to compute
  - many other algorithms exists, but not covered here

© 2020 Fabio Pellacini

Path tracing – informal

[Disney/Youtube]

© 2020 Fabio Pellacini •

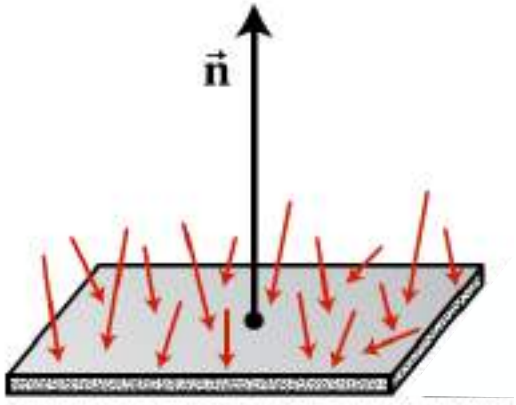
Rendering Equation

© 2020 Fabio Pellacini

Radiant power

- Energy *Q* [J]: illumination energy
  - what a detector measures
  - intuition: how many photons hit a detector
- Power *P* or flux [W=Js^{-1}]: energy per unit time
  - intuition: how many photons hit a surface per second
  - other units derived from this

$$P = \frac{dQ}{dt}$$



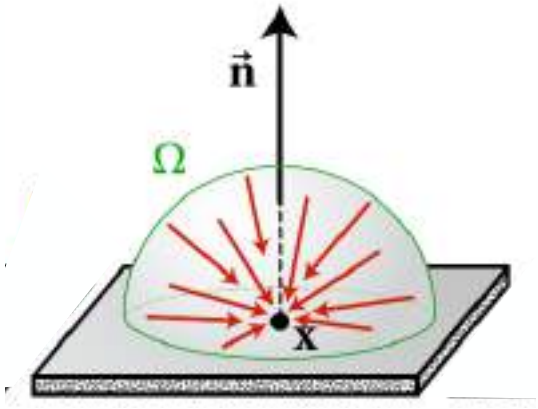
[Woiciech Jaroz]

© 2020 Fabio Pellacini

Irradiance

- Irradiance *E* [Wm^{-2}]: power per unit area
  - measured at each location on a surface
  - intuition: considers photons coming from all direction to a small area around a single surface point
  - irradiance is used for incoming light
  - radiosity is the term used for outgoing light

$$E(\mathbf{x}) = \frac{dP}{dA_{\mathbf{x}}}$$

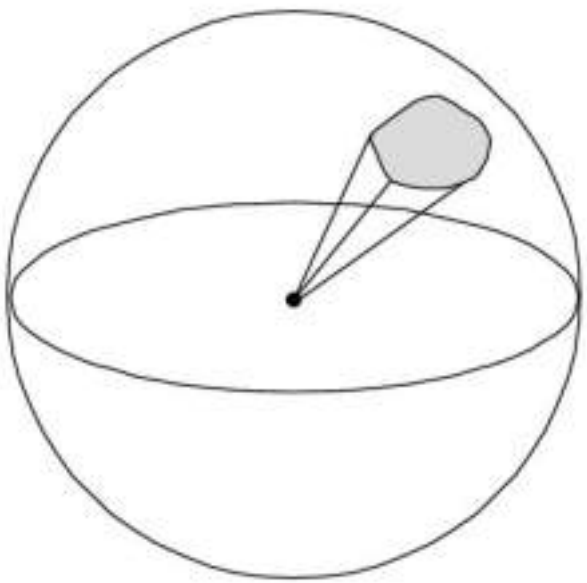


[Woiciech Jaroz]

© 2020 Fabio Pellacini

Solid Angle

- Solid angle [sr]: area of the unit sphere subtended by an object
  - analogous to angles in 2D that are the arc length of the unit circle
  - intuition: measure the "size" of a light beam
- Differential solid angle [sr]: infinitesimal solid angle around a direction
  - for the direction **d**, indicated as $d\omega_{\mathbf{d}}$



[Jaroslav Krivanec]

© 2020 Fabio Pellacini

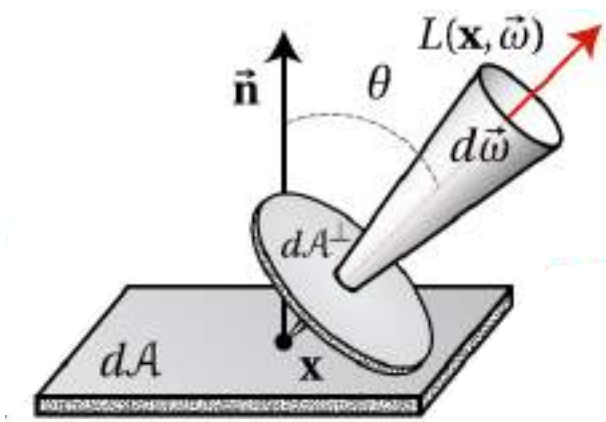
Radiance

- Radiance L [$Wm^{-2}sr^{-2}$]: power per unit projected area per unit solid angle
  - measured at each surface location and for each direction
  - intuition: consider only light in a thin beam around a direction
  - "projected area": account for orientation of surface
  - radiance is defined for incoming, outgoing and reflected light
  - most important quantity: *what our eyes see*
  - HDRs store radiance (scaled)

$$L(\mathbf{x}, \mathbf{i}) = \frac{d^2 P}{dA_{\mathbf{x}} d\omega_{\mathbf{i}} (\mathbf{i} \cdot \mathbf{n})}$$

$$dA_{\mathbf{x}}^{\perp} = dA(\mathbf{i} \cdot \mathbf{n})$$

$$d\omega_{\mathbf{i}}^{\perp} = d\omega_{\mathbf{i}}(\mathbf{i} \cdot \mathbf{n})$$



[Woiciech Jaroz]

© 2020 Fabio Pellacini

Radiance

- Typical values [cd m^{-2}]
  - surface of the sun: 2,000,000,000
  - cloudy sky: 30,000
  - clear day: 3,000
  - overcast day: 300
  - surface of the moon: 0.03

[Pat Hanrahan]

© 2020 Fabio Pellacini

Radiance properties

- Radiance depends on the wavelength
  - we just use RGB vector to represent it that samples the spectrum at three wavelength
  - good enough for most cases

$$L(\mathbf{x}, \mathbf{o}, \lambda) \approx [L_R(\mathbf{x}, \mathbf{o}), L_G(\mathbf{x}, \mathbf{o}), L_B(\mathbf{x}, \mathbf{o})]^T$$

- Incoming radiance L_i is not the same as outgoing radiance L_o
  - photons that reach the surface are not the same as the ones that leave it
  - if not specified, we mean outgoing radiance

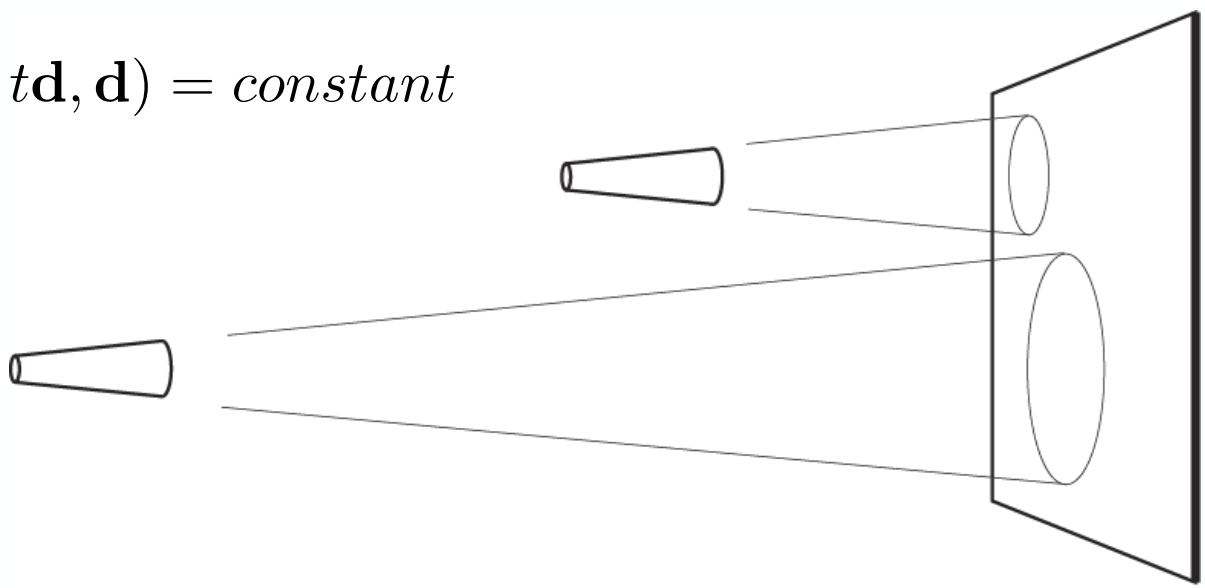
$$L_o(\mathbf{x}, \mathbf{d}) = L(\mathbf{x} \rightarrow \mathbf{d}) \neq L(\mathbf{x} \leftarrow \mathbf{d}) = L_i(\mathbf{x}, \mathbf{d})$$

© 2020 Fabio Pellacini

Radiance properties

- Radiance does not change along straight lines in empty space
  - comes from conservation of energy
  - approximately true also in air for short distances
  - reason why radiance is associated with rays in a raytracer
  - corollary: surface colors do not change when we move away

$$L(\mathbf{o} + t\mathbf{d}, \mathbf{d}) = constant$$



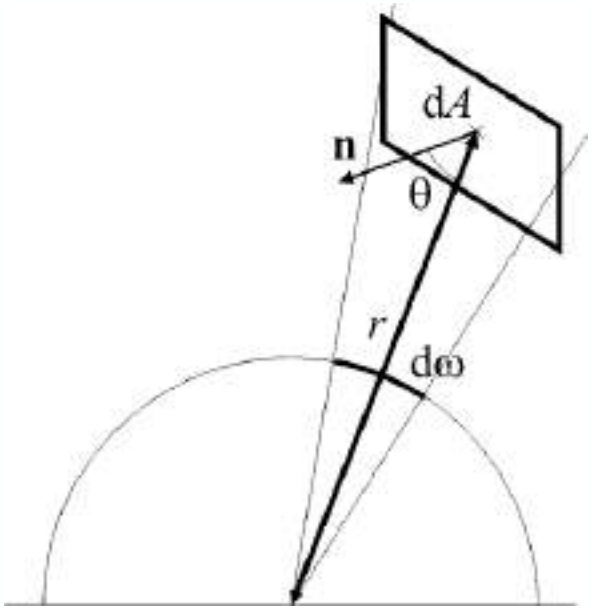
[Shirley]

© 2020 Fabio Pellacini

Differential solid angle of surface

- Differential solid angle of a surface
  - consider a small surface area *dA*
  - project it onto the unit sphere by dividing by distance squared r^2
  - take into account orientation by correcting with the cosine

$$d\omega = \frac{dA \cos\theta}{r^2}$$



[Jaroslav Krivanec]

© 2020 Fabio Pellacini

Proof of radiance invariance

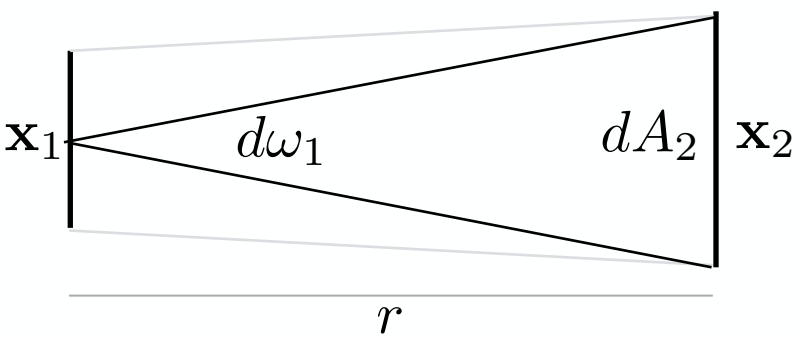
- Consider two surfaces orthogonal to each other
- For energy conservation all photons leaving a surface within the solid angle of the other have to reach the other surface

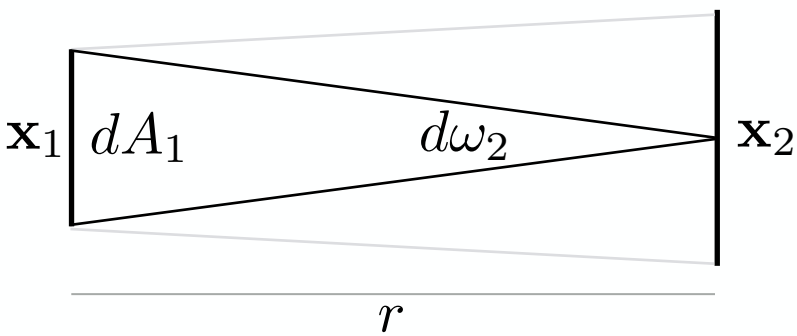
$$d^2P_1 = d^2P_2 \qquad \text{energy conservation}$$

$$L_1 d\omega_1 dA_1 = L_2 d\omega_2 dA_2 \qquad \text{by substitution}$$

$$L_1(dA_2/r^2)dA_1 = L_2(dA_1/r^2)dA_2 \qquad \text{by construction}$$

$$L_1 = L_2 = d^2P\frac{dA_1 dA_2}{r^2} \qquad \text{invariance along rays}$$

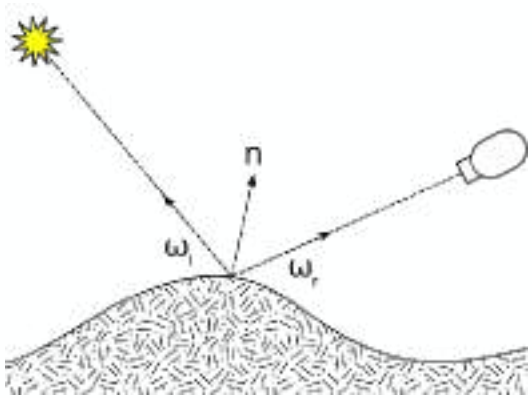


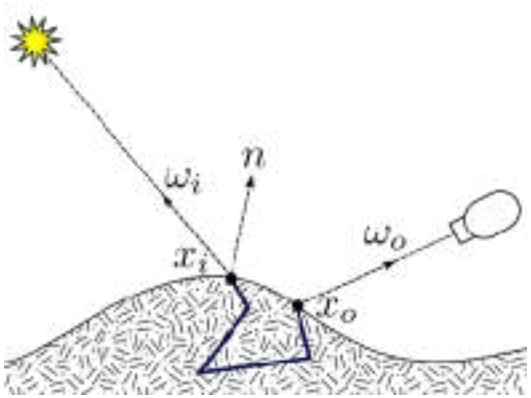


© 2020 Fabio Pellacini

Scattering

- Photons that come to a surface from one direction are scattered in many other directions
- In graphics, we consider mostly three main scattering "modes"
- *Surface scattering* at the surface, e.g. metals and dielectrics
- *Subsurface scattering* in a thin region right below the surface, e.g. skin
- *Volumetric scattering* in a 3D volume, e.g. smoke and fog
- We'll focus on surface scattering



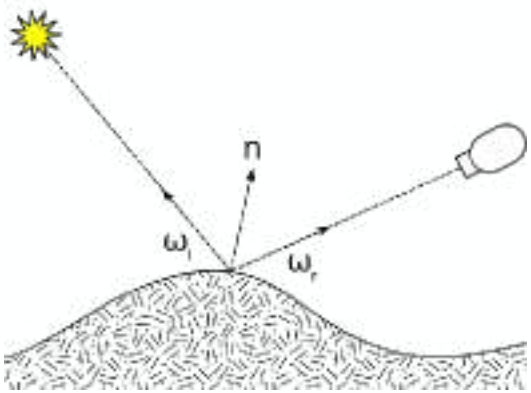


© 2020 Fabio Pellacini

Surface scattering: BRDF

- BRDF [sr^{-1}]: *bidirectional reflectance distribution function*
- Defined as the ratio of the differential reflected radiance over the differential incident irradiance
- Depends on incoming and outgoing directions, and wavelength of light
  - do not track wavelength explicitly: just use colors for BRDF coeff.
- Intuition: probability density that a photon from a direction *i* of solid angle $d\omega_i$ is reflected in direction *o* of solid angle $d\omega_o$

$$f(\mathbf{x}, \mathbf{i}, \mathbf{o}) = \frac{dL_r(\mathbf{x}, \mathbf{o})}{dE_i(\mathbf{x}, \mathbf{i})} = \frac{dL_r(\mathbf{x}, \mathbf{o})}{L_i(\mathbf{x}, \mathbf{i})(\mathbf{n_x} \cdot \mathbf{i})d\omega_\mathbf{i}}$$



© 2020 Fabio Pellacini

Reflected radiance with BRDF

- From the BRDF definition, we write the differential reflected radiance as the product of the BRDF and the incoming radiance weighted by the cosine
  - write the point explicitly

$$dL_r(\mathbf{x}, \mathbf{o}) = L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$

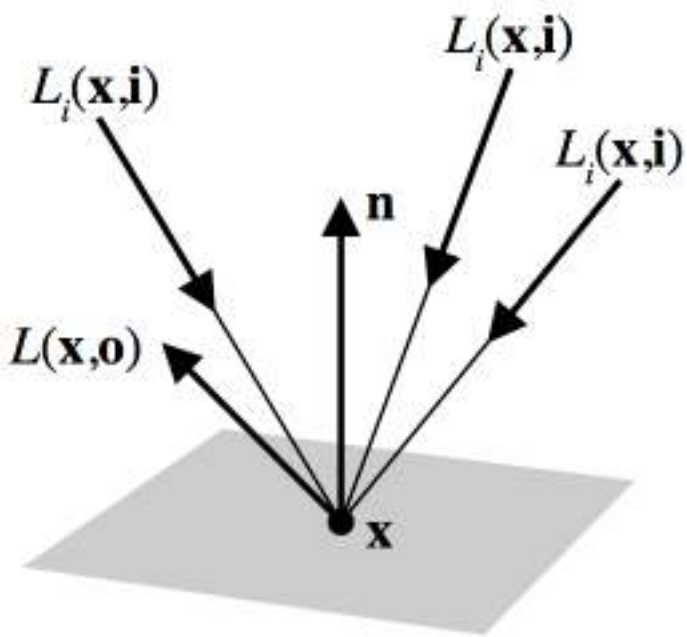
- The total reflected radiance is the integral of the differential reflected radiance for all directions over the hemisphere

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H_\mathbf{x}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$

© 2020 Fabio Pellacini

Reflected radiance

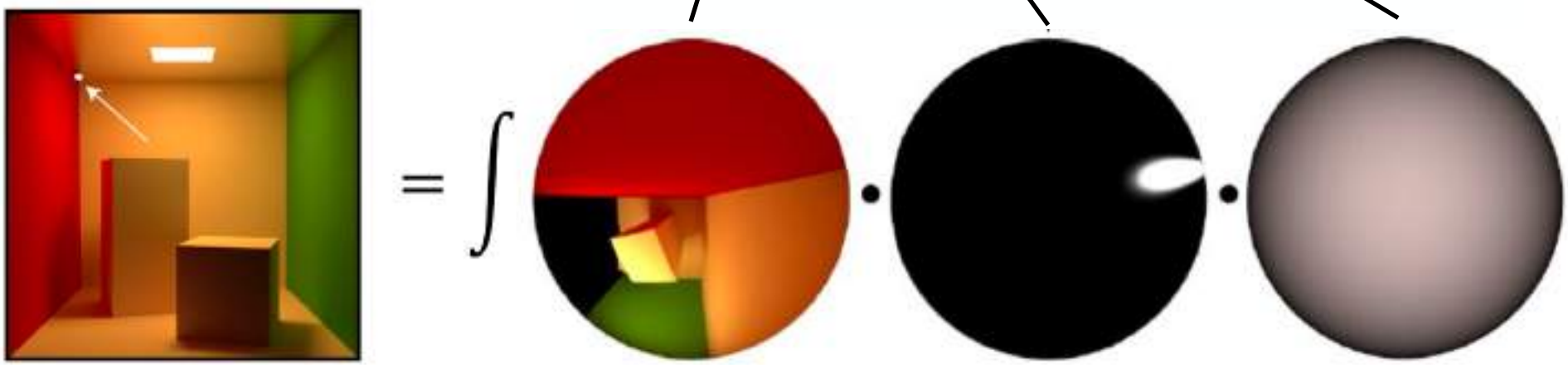
$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H_\mathbf{x}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$



© 2020 Fabio Pellacini

Reflected radiance

$$L_r(\mathbf{x},\mathbf{o}) = \int_{H_\mathbf{x}} L_i(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o})(\mathbf{n_x}\cdot\mathbf{i})d\omega_\mathbf{i}$$



[Krivanec]

© 2020 Fabio Pellacini

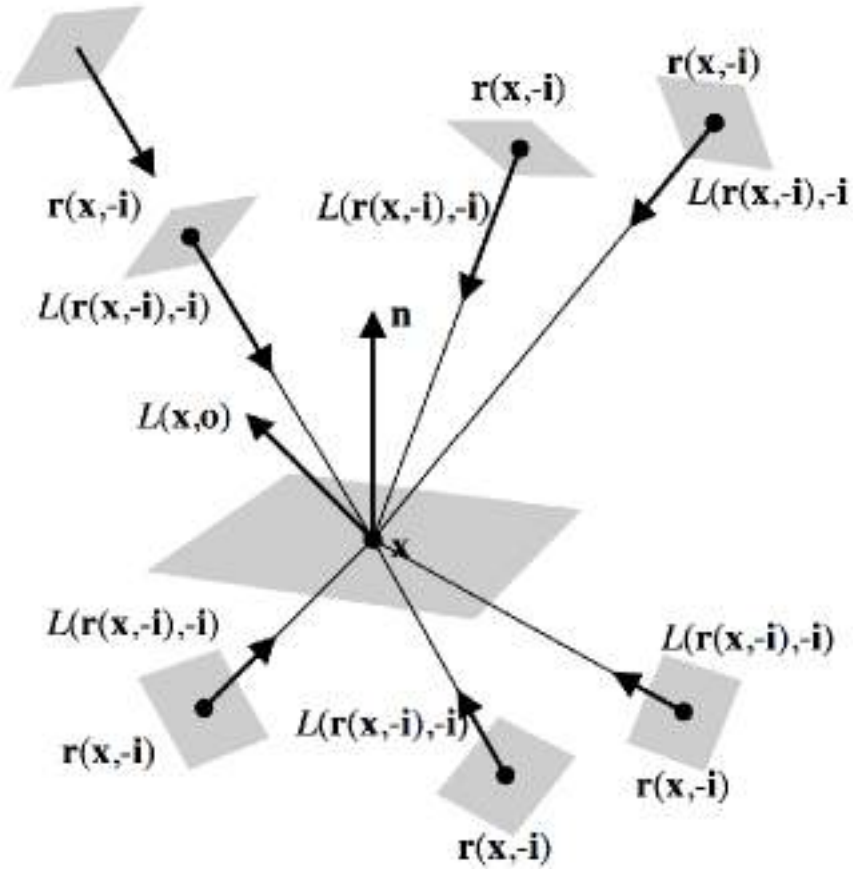
BSDF

- So far we only discussed opaque objects
- For translucent objects, such as glass, we have similar definitions
- BTDF: *bidirectional transmittance distribution function*
  - equiv. to BRDF but bottom hemisphere
- BSDF: *bidirectional scattering distribution function*
  - considers both top and bottom hemisphere
  - sum of BRDF and BTDF
- BSDF reflection integral is over full sphere around a point and takes the absolute value of the cosine

© 2020 Fabio Pellacini

Reflected radiance with BSDF

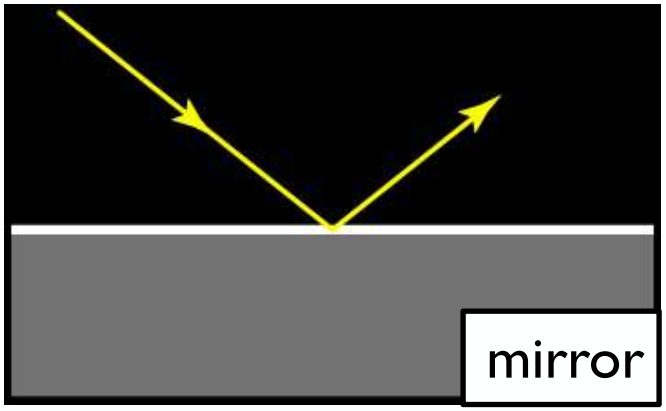
$$L_r(\mathbf{x},\mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

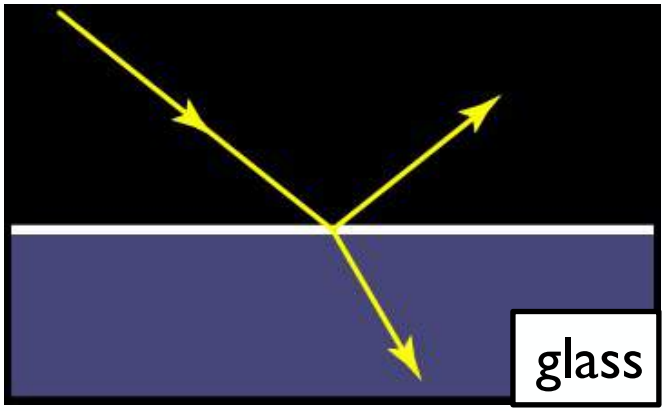


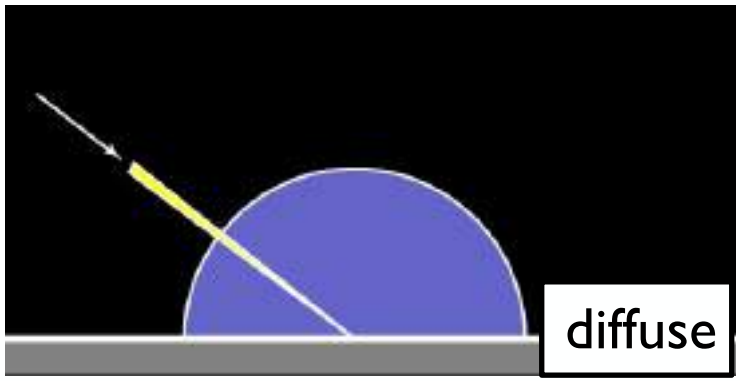
© 2020 Fabio Pellacini

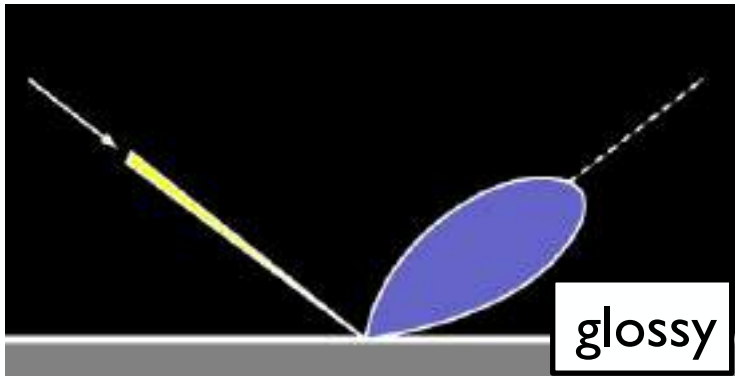
BSDFs for different materials

- Different materials are described by different scattering distributions









© 2020 Fabio Pellacini

Properties of the BSDF

- Positivity: since photons cannot be "negatively" reflected

$$f(\mathbf{x}, \mathbf{i}, \mathbf{o}) \geq 0$$

- Helmotz reciprocity: can invert light paths

$$f(\mathbf{x}, \mathbf{i}, \mathbf{o}) = f(\mathbf{x}, \mathbf{o}, \mathbf{i})$$

- Energy conservation: all photons that comes in are either reflected or absorbed and no new photons is emitted

$$\forall \mathbf{i}. \int_{H^2} f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{o} \cdot \mathbf{n}| d\omega_{\mathbf{o}} \leq 1$$

© 2020 Fabio Pellacini

Rendering equation

- Start from the equation for reflected radiance

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- Consider also the light that is naturally emitted by some surfaces, such as light source with an additional term

$$L_e(\mathbf{x}, \mathbf{o})$$

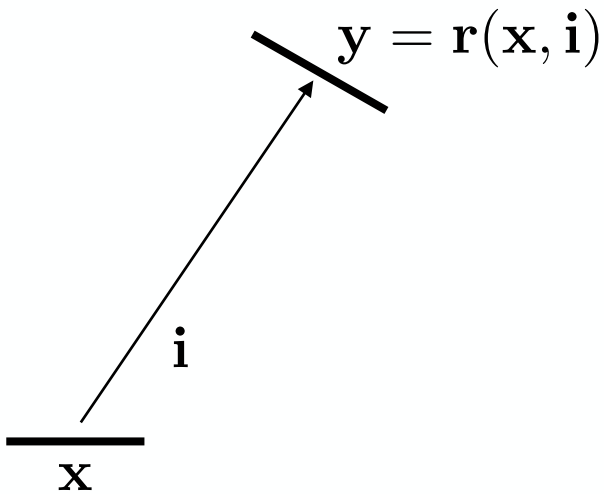
- Now we want to express the incoming radiance w.r.t. the outgoing radiance, using the property that radiance along a ray does not change

© 2020 Fabio Pellacini

Radiance along a ray

- From the invariance of radiance along a ray, we can express the radiance incoming to a point as the radiance leaving other points
- Express this with recasting to find the first visible point

$$L_i(\mathbf{x}, \mathbf{i}) = L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i}))$$



© 2020 Fabio Pellacini

Rendering equation

- Start from the equation for reflected radiance

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- Write incoming as the outgoing radiance os the first visible surface along the ray

$$L_i(\mathbf{x}, \mathbf{i}) = L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i})$$

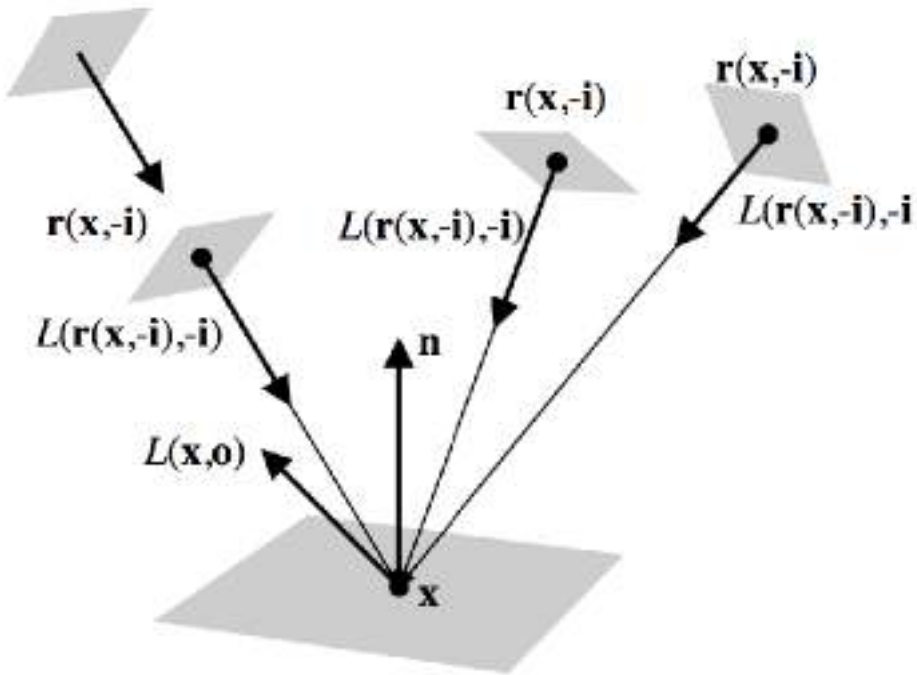
- Substitute the latter in the former and add the emitted radiance

$$L(\mathbf{x}, \mathbf{o}) = L_e(\mathbf{x}, \mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

© 2020 Fabio Pellacini

Rendering equation

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

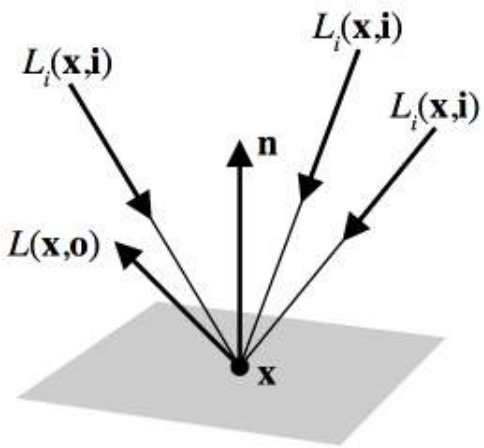


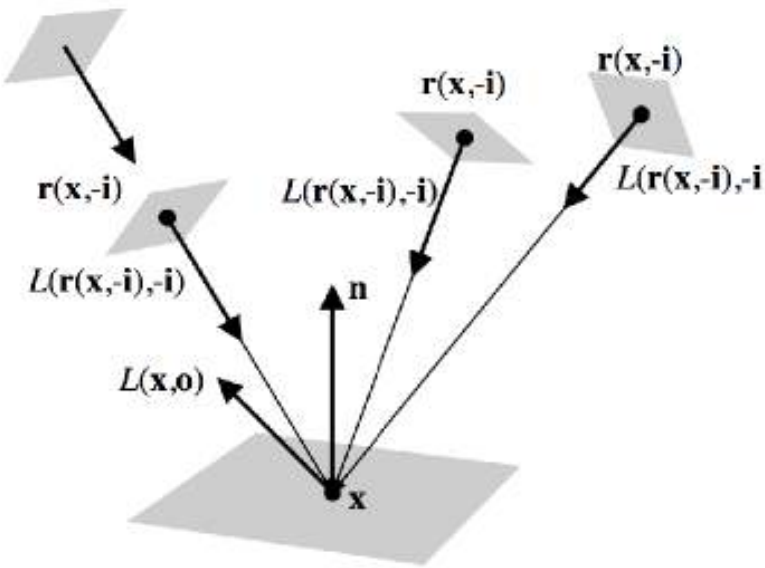
© 2020 Fabio Pellacini

Reflection vs rendering equation

$$L_r(\mathbf{x},\mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$





© 2020 Fabio Pellacini

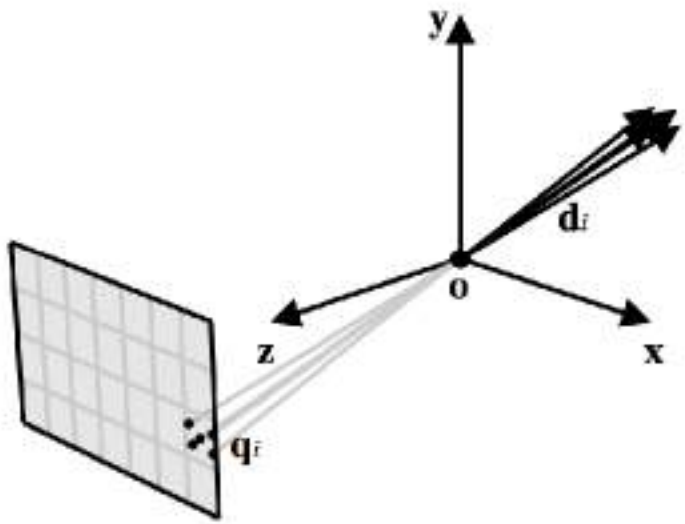
Rendering vs reflection equation

- **Reflection equation**: describes *local light reflection* at a point
  - *Integral* can be used to compute the reflected radiance given the *known incoming radiance*
- **Rendering equation**: condition of the global distribution of light
  - describe the energy balance in the scene at steady state
  - *Integral equation* with *unknown outgoing radiance* on both sides
- **Rendering**: compute the outgoing radiance for all visible points in the images by solving the rendering equation for those points and their view directions

© 2020 Fabio Pellacini

Pixel radiance

- All previous formulation specify the radiance at a point in the scene
- To compute the pixel radiance we can think of measuring the incoming radiance in the sensor area covered by the pixel that can reach the pixel through the lens aperture
- Let us consider first the case of a pinhole camera depicted here



© 2020 Fabio Pellacini

Pixel radiance

- We measure the pixel radiance as the integral, over the pixel surface, of the incoming radiance to each product weighted by the sensitivity of the sensor *s* and the cosine; this gives us proper anti-aliasing
- To be independent of the image resolution we divide by the pixel area

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\mathbf{q} \in pixel} L_i(\mathbf{q}, \mathbf{d}) s(\mathbf{q}, \mathbf{d}) |\mathbf{d} \cdot \mathbf{n_q}| dA_{\mathbf{q}} \text{ with } \mathbf{d} = \frac{\mathbf{e} - \mathbf{q}}{|\mathbf{e} - \mathbf{q}|}$$

- We now make the approximation that sensor has constant sensitivity and corrects for the cosine (this is your vignetting by the way)

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\mathbf{q} \in pixel} L_i(\mathbf{q}, \mathbf{d}) dA_{\mathbf{q}} \text{ with } \mathbf{d} = \frac{\mathbf{e} - \mathbf{q}}{|\mathbf{e} - \mathbf{q}|}$$

© 2020 Fabio Pellacini

Monte Carlo integration

© 2020 Fabio Pellacini

Discreet random variable

- Given a random processes with a finite number of outcomes
- A *discreet random variable* is a function that maps outcomes to measurable values, typically real numbers

$$X : \Omega \rightarrow D$$

- Example: rolling a dice
  - outcomes: the face of the dice
  - values of the variable: numbers 1, 2, 3, 4, 5, 6
- For simplicity in these notes, we consider just the range of the random variable and discuss its property over that
  - we assume there is a proper random process whose outcomes can be mapped in the range
- Disclaimer: next slides have math that is correct but not as formal!

© 2020 Fabio Pellacini

Discreet random variable

- A *discreet random variable* is a variable that can randomly take one of a finite number of values with fixed probabilities

$$X \in D = \{v_i\}$$

- *Probability mass function*: probability of each value

$$X \sim p_X \quad p(x) = \Pr(X = x) \quad \sum_i p(v_i) = 1$$

- *Cumulative distribution function*: probability to be within an interval

$$c(x) = \Pr(X \leq x) \quad c(x) = \sum_{v_i \leq x} p(v_i) \quad c(\max(v_i)) = 1$$

- Example: rolling a dice

$$p(v_i) = 1/6 \qquad v_i = \{1, 2, 3, 4, 5, 6\}$$

© 2020 Fabio Pellacini

Expected value and variance

- Expected value: weighted average of the variable values

$$E[X] = \sum_i v_i p(v_i)$$

- Variance: the expected value of the difference from the average

$$\sigma^2[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

- Example: expected value of rolling a dice

$$E[X] = (1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5$$

$$\sigma^2[X] = ... \approx 2.92$$

© 2020 Fabio Pellacini

Estimating expected value

- Estimate expected value by running an experiment
  - pick at random one of the variable values with probably *p*
  - lets call this value x_i
  - repeat *n* times and average the results

$$E[X] \approx \frac{1}{N} \sum_i^N x_i$$

- Larger *n* give better estimates
- Example: rolling a dice
  - roll 3 times: $\{3, 1, 6\} \rightarrow E[X] \approx (3 + 1 + 6)/3 = 3.33$
  - roll 9 times: $\{3, 1, 6, 2, 5, 3, 4, 6, 2\} \rightarrow E[X] \approx 3.51$

© 2020 Fabio Pellacini

Law of large numbers

- As the number of trials goes to infinity, the probability that the average opt the observations is equal to the expected value will be equal to one.

$$\Pr\left(E[X] = \lim_{N\to\infty} \frac{1}{N} \sum\nolimits_i x_i\right) = 1$$

- This means that *the estimate of the expected value converges to the correct value*

© 2020 Fabio Pellacini

Continuous random variable

- *One-dimensional continuous random variables*: similar to discreet random variables but the values are in the continuous domain

$$X \in D = [a, b)$$

- *Probability density function (pdf)*: probability that the variable is in within a differential interval around a given value

$$X \sim p_X \quad p(x)dx = d\Pr(x \leq X \leq x + dx) \quad \int_a^b p(x)dx = 1$$

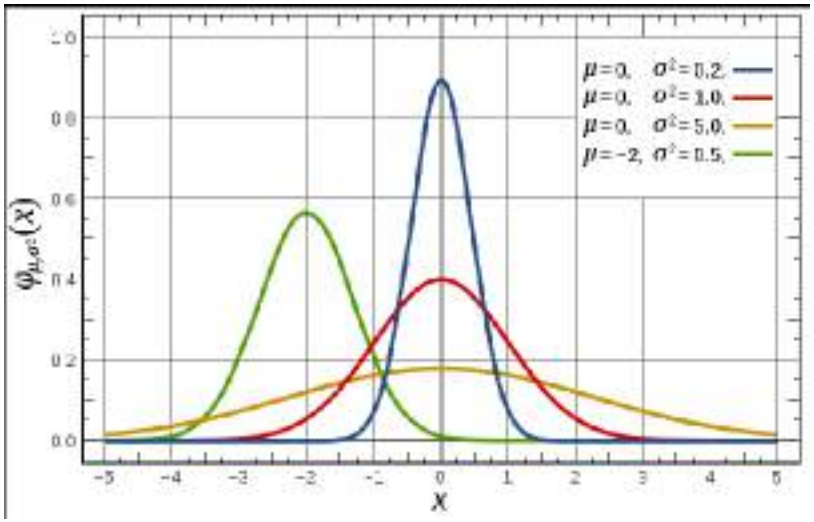
- *Cumulative distribution function*: probability to be within an interval

$$c(x) = \Pr(X \leq x) \quad c(x) = \int_a^x p(y)dy \quad c(b) = 1$$

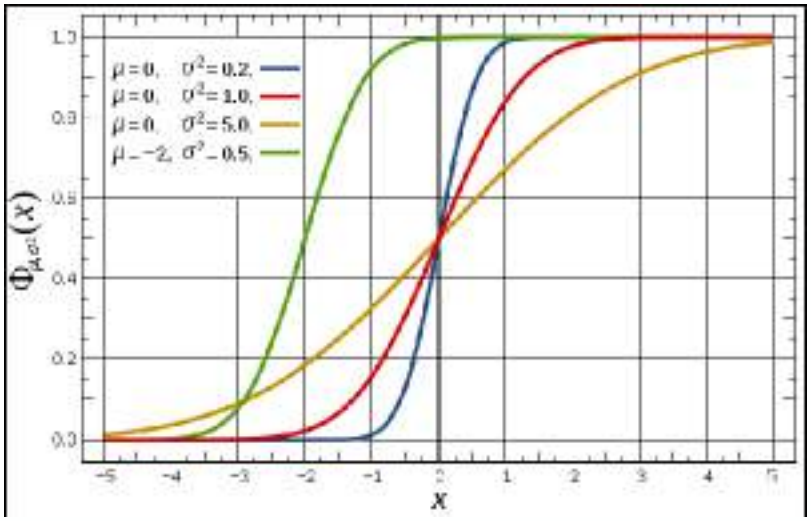
© 2020 Fabio Pellacini

Continuous random variable

example pdf: gaussian *p(x)*



example cdf: erf *c(x)*



© 2020 Fabio Pellacini

Continuous random variable

- More generally, a *continuous random variable* is a multi-dimensional variable that can take a value in a arbitrary continuous domain

$$\mathbf{X} \in D$$

- *Probability density function*: probability that the variable takes a value within a differential domain around a point

$$\mathbf{X} \sim p_{\mathbf{X}} \quad p(\mathbf{x})dA_{\mathbf{x}} = d\Pr(\mathbf{x} \in dD_{\mathbf{x}}) \qquad \int_D p(\mathbf{x})dA_{\mathbf{x}} = 1$$

- *Cumulative distribution function*: general formulation is complex
- Expected values and variance

$$E[\mathbf{X}] = \int_{\mathbf{x}\in D} \mathbf{x}p(\mathbf{x})dA_{\mathbf{x}} \quad \sigma[\mathbf{X}] = \int_{\mathbf{x}\in D} |\mathbf{x} - E[\mathbf{X}]|^2 p(\mathbf{x})dA_{\mathbf{x}}$$

© 2020 Fabio Pellacini

Expected value of a function

- Expected value of a function of a continuous random variable

$$E[g(\mathbf{X})] = \int_{\mathbf{x} \in D} g(\mathbf{x}) p(\mathbf{x}) dA_{\mathbf{x}}$$

- Variance

$$\sigma^2[g(\mathbf{X})] = E[g(\mathbf{X}) - E[g(\mathbf{X})]]^2 = \int_{\mathbf{x} \in D} (g(\mathbf{x}) - E[g(\mathbf{X}])) ^2 p(\mathbf{x}) dA_{\mathbf{x}}$$

- Estimated expected value

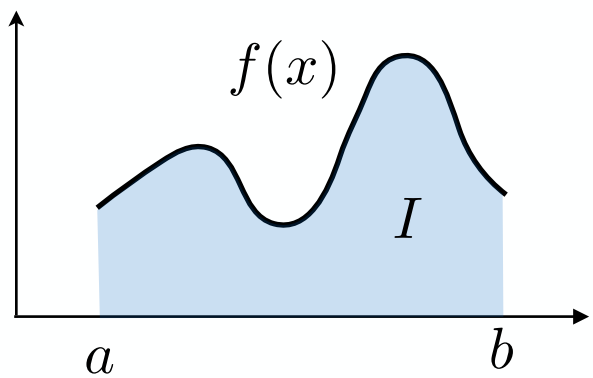
$$E[g(\mathbf{X})] \approx \frac{1}{N} \sum\nolimits_i g(\mathbf{x}_i)$$

© 2020 Fabio Pellacini

Numerical integration

- Rendering amounts to solving numerical integrals
- Let us consider first the case of one dimensional integrals
  - in this case, the integral is the area below the curve

$$I = \int_a^b f(x)dx$$



© 2020 Fabio Pellacini

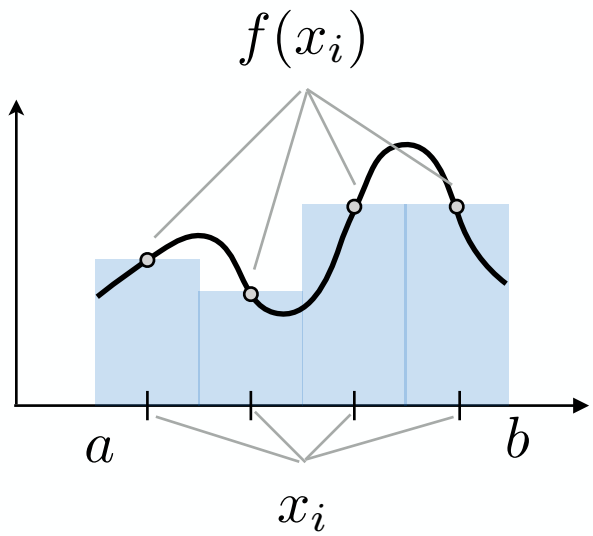
Numerical quadrature

- The classic method for solving numerical integrals is to sample the function at *n* equal spaces positions and approximate the integral with the corresponding boxes

$$I = \int_a^b f(x)dx$$

$$x_i = a + (i - 0.5) \cdot \frac{b-a}{N}$$

$$I \approx \sum_{i=1}^{N} f(x_i) \cdot \frac{b-a}{N}$$



© 2020 Fabio Pellacini

Monte Carlo integration

we want to evaluate $$I = \int_a^b f(x)dx$$

by definition $$E[g(X)] = \int_a^b g(x)p(x)dx$$

can be estimated as $$E[g(X)] \approx \frac{1}{n}\sum_{i=0}^{n-1} g(x_i)$$

by setting $$g(x) = \frac{f(x)}{p(x)} \text{ with uniform } p(x) = \frac{1}{b-a}$$

we have that $$I = \int_a^b \frac{f(x)}{p(x)} p(x)dx$$

can be evaluated as $$I \approx \frac{1}{n}\sum_{i}^{n} \frac{f(x_i)}{p(x_i)}$$

© 2020 Fabio Pellacini

Monte Carlo integration

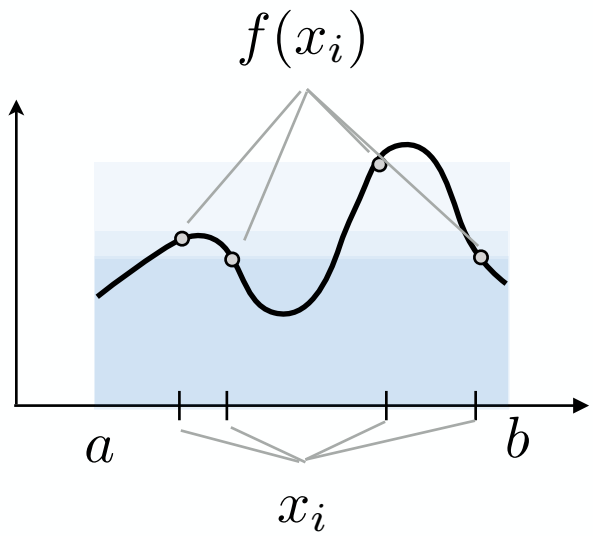
- Approximately evaluate the integral using a *uniform random variable*, by averaging function values at random positions in [a,b)

$$I = \int_a^b f(x)dx$$

$$x \sim p(x) = \frac{1}{b-a}$$

$$I \approx \frac{1}{N} \sum_i \frac{f(x_i)}{p(x_i)} =$$

$$= \frac{1}{N} \sum_i f(x_i)(b-a)$$



© 2020 Fabio Pellacini

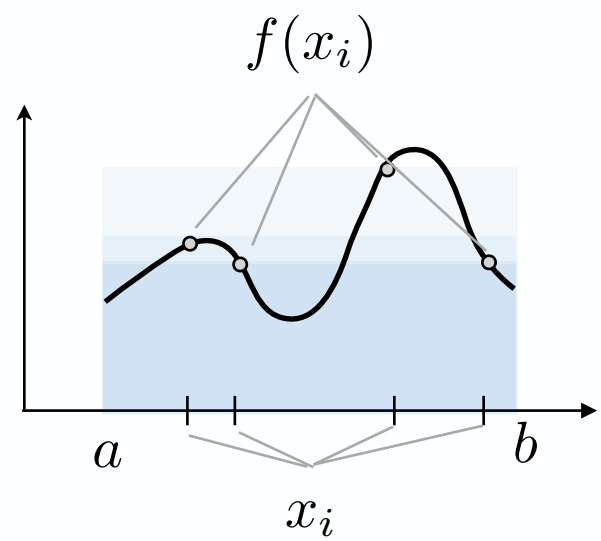
Monte Carlo integration

- Approximately evaluate the integral using a *uniform random variable*, by averaging function values at random positions in [a,b)

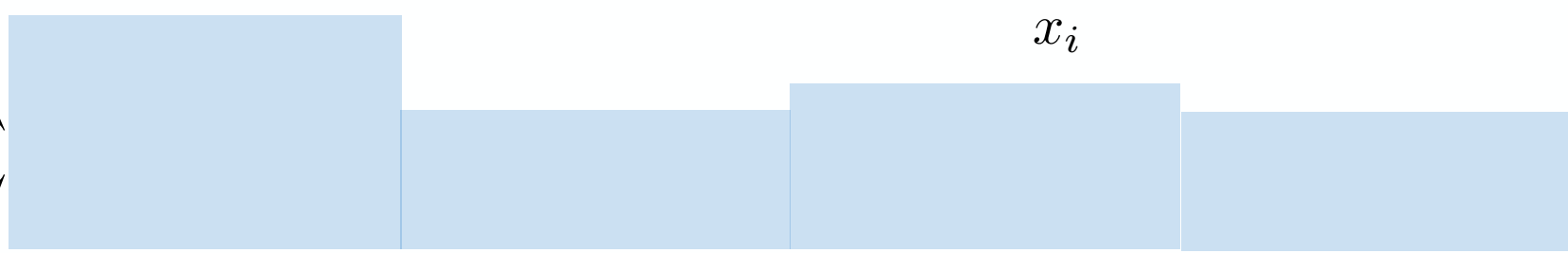
$$I = \int_a^b f(x)dx$$

$$x \sim p(x) = \frac{1}{b-a}$$

$$I \approx \frac{1}{N} \sum_i f(x_i)(b-a)$$



$$I \approx \frac{1}{N} \sum$$



© 2020 Fabio Pellacini

Monte Carlo vs. quadrature

- Evaluate the same integral

$$I = \int_a^b f(x)dx$$

- with stochastic vs deterministic positions

$$x \sim p(x) = \frac{1}{b-a} \qquad\qquad x_i = a + (i + 0.5) \cdot \frac{b-a}{n}$$

- using averaging vs reconstruction

$$I \approx \frac{1}{N} \sum_i f(x_i)(b-a) \qquad\qquad I \approx \sum_{i=1}^{N} f(x_i) \cdot \frac{b-a}{N}$$

© 2020 Fabio Pellacini

Curse of dimensionality

- Evaluate an integral of a function of *k* variables, here in a "box" domain

$$I = \int_{\mathbf{x} \in D} f(\mathbf{x}) dA_{\mathbf{x}} = \int_{a_1}^{b_1} \int_{a_2}^{b_2} ... \int_{a_k}^{b_k} f(x_1, x_2, ..., x_k) dx_1 dx_2 ... dx_k$$

- With numerical quadrature we split each axis separately
- The integral is then approximated by summing the function values of the n^k *k*-dimensional intervals

$$I \approx \sum_{i_1}^{N_1} \sum_{i_2}^{N_2} ... \sum_{i_k}^{N_k} f(x_{1,i_1}, x_{2,i_2}, ..., x_{k,i_k}) w_1 w_2 ... w_k$$

- Error in the estimate goes down as

$$error \propto \frac{1}{\sqrt[k]{N}}$$

© 2020 Fabio Pellacini

Monte Carlo integration

- The estimation of the expected values works in *any dimension*
  - just pick random numbers in the proper high dimensional domain

$$I = \int_{\mathbf{x} \in D} f(\mathbf{x}) dA_{\mathbf{x}} \approx \frac{1}{N} \sum_i \frac{f(\mathbf{x})}{p(\mathbf{x})}$$

- Works for any shape of the domain, provided that you can picks random numbers in it
- The expected relative error of the random estimate depends only on the number of samples, and not the dimensionality of the domain, and goes down with the square root of *n*

$$error \propto \frac{1}{\sqrt{N}}$$

© 2020 Fabio Pellacini

Example: computing *pi*

- *Pi* can be evaluated by computing the integral of a function that is one inside a unit circle and 0 outside

$$\pi = \int_{-1}^{1}\int_{-1}^{1} f(x,y)dxdy \text{ with } f(x,y) = \begin{cases} 1 & \text{if } x^2+y^2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

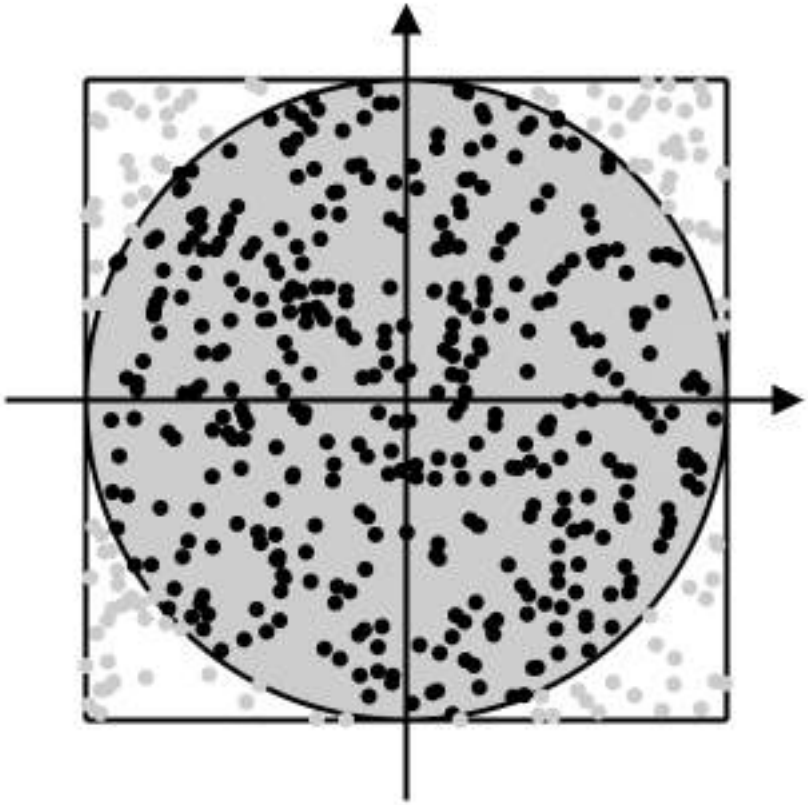
- With Monte Carlo integration, we can estimate *pi* as

$$\pi \approx \frac{4}{N}\sum_{i}^{N} f(x_i,y_i) \text{ with } (x_i,y_i) \in [-1,1]^2, p(x_i,y_i) = \frac{1}{4}$$

© 2020 Fabio Pellacini

Example: computing *pi*

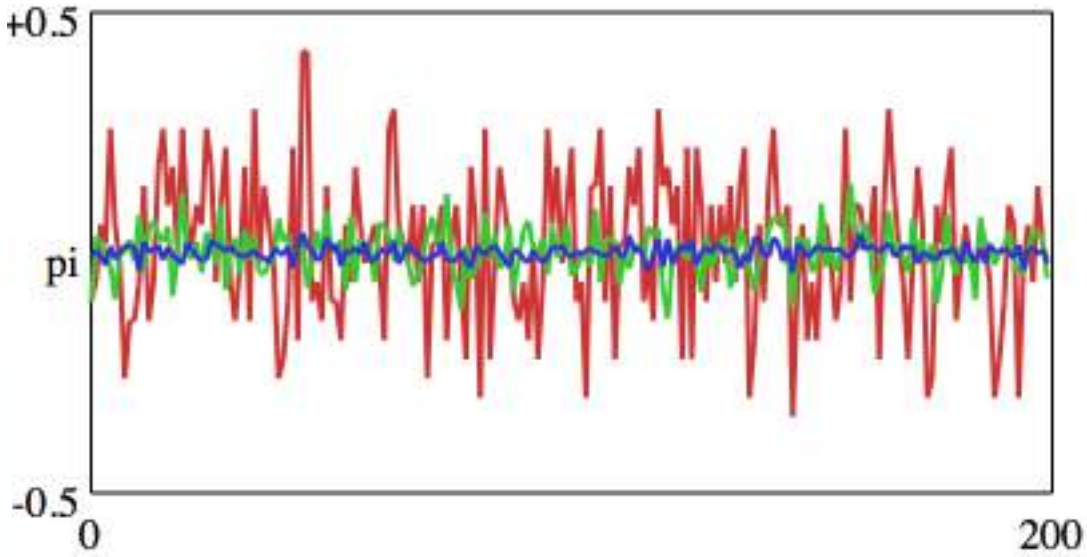
- If we run one experiment we obtain the following image



© 2020 Fabio Pellacini

Example: computing *pi*

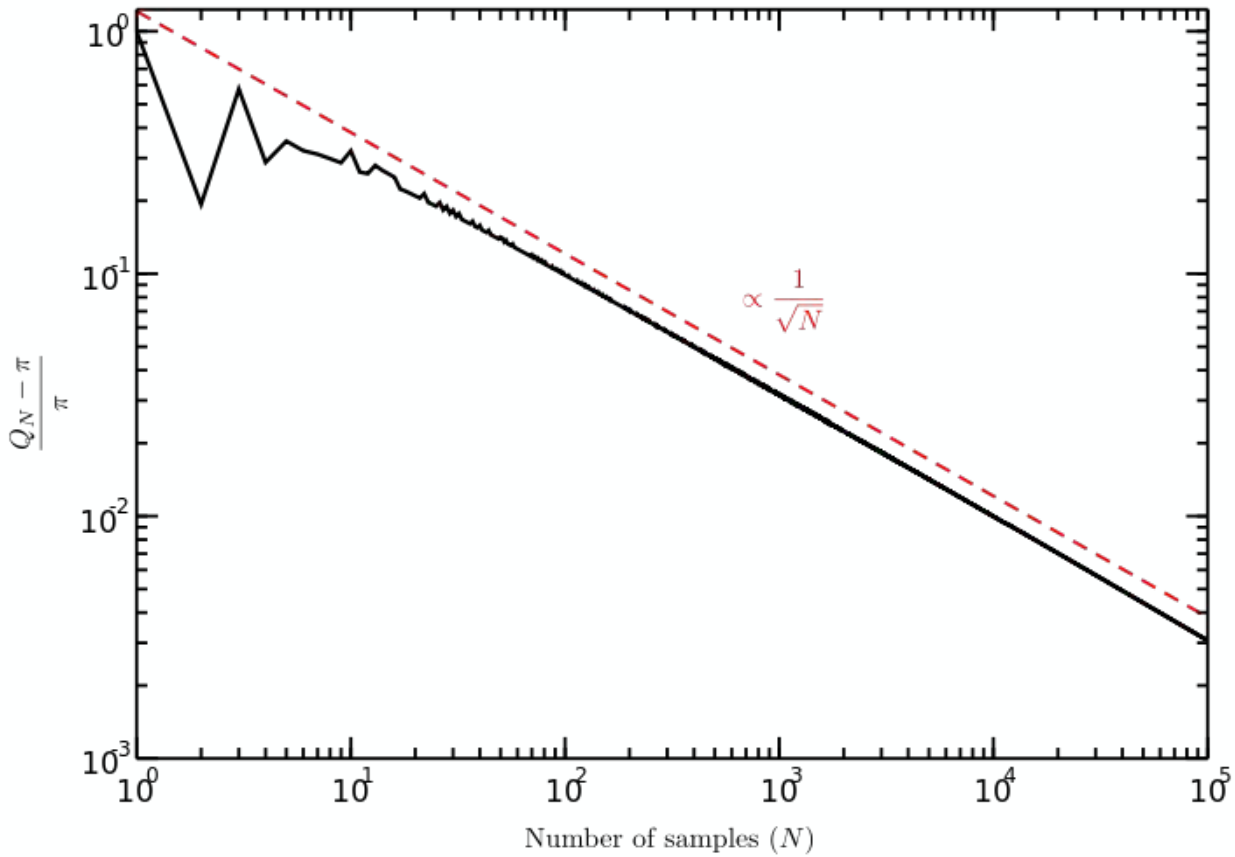
- Each time we run an experiment we get a slightly different value of *pi*
- The estimates get better as *N* increases



© 2020 Fabio Pellacini

Example: computing *pi*

- Variance decreases with the square root of *N*



[Wikipedia]

© 2020 Fabio Pellacini

Generating random numbers

- Monte Carlo methods requires random numbers
- In general, we use *pseudo random numbers*, i.e. numbers that are deterministically generated, but appear random
  - true random numbers are not always available and are expensive
- Many generators exists with different tradeoffs between random quality and speed/memory
- All generators produce sequences pseudo-random bits, that can be used to generate floating point values in the [0,1) range
- I particularly like the PCG family of random numbers by Melissa E. O'Neill, since they are simple, have good statistical properties, are efficient and they allow for multiple independent streams of random numbers

© 2020 Fabio Pellacini

Implementing the *pi* estimate

- Now we have a way to generate uniform random floats in [0,1)
- But we need uniform values in [-1,1]x[-1,1]
- To generate random numbers in higher dimension, we simply generate multiple random numbers with our generators since the numbers are statistically independent
  - the pdf is the product of the one-dimensional pdfs
- To generate random numbers in a different domain, we *warp* the random numbers to that domain; in out case this is just scaling
  - the pdf needs to be appropriately "rescaled" to take into account the new domain size and the change of variable
- If *r* is our generator of uniform random numbers in [0,1) we can write

$$r \sim p(r) = 1 \rightarrow \mathbf{x} = (2r_1 - 1, 2r_2 - 1) \sim p(\mathbf{x}) = 1/4$$

© 2020 Fabio Pellacini

Implementing the *pi* estimate

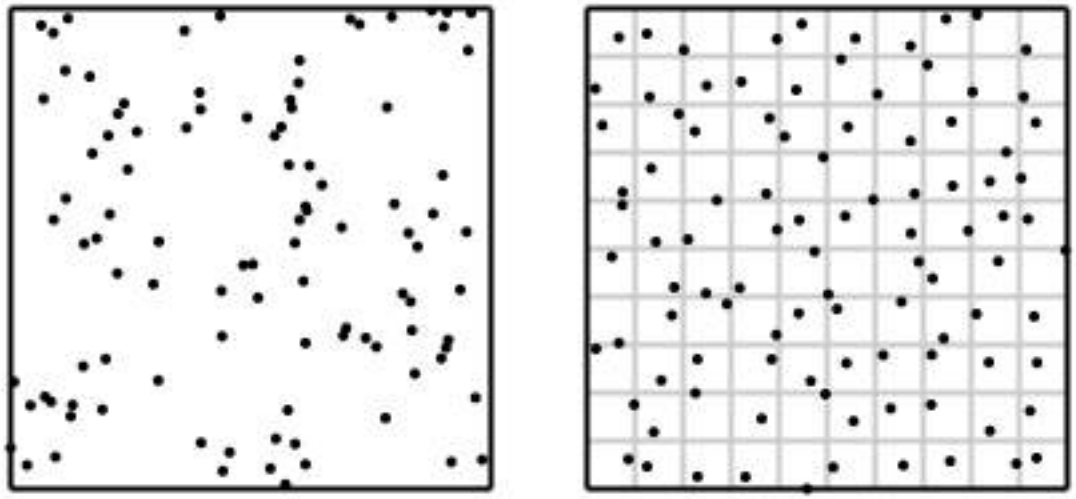
```
float f(vec2f x) {
    return (x.x * x.x + x.y * x.y < 1) ? 1 : 0;
}

float pi(int nsamples, rng_state& rng) {
    auto integral = 0.0f;
    for (auto s = 0; s < nsamples; s++) {
       integral += f({randf(rng)*2-1,randf(rng)*2-1});
    }
    integral *= 4.0f / nsamples;
    return integral;
}
```


© 2020 Fabio Pellacini

Stratified sampling

- Main issue with Monte Carlo is the noise in the solution of integrals
- Several methods exists to reduce noise, which amount to choosing samples at better locations, but still randomly
- *Stratified sampling:* avoid "clumps" in uniform random numbers by forcing them to be distributed in smaller domains
  - not so convenient to implement in rendering

$$\mathbf{x} = \left(\frac{i + r_1}{N_1}, \frac{j + r_2}{N_2} \right)$$



© 2020 Fabio Pellacini

Importance sampling

- *Importance sampling:* sample with a pdf that is as close as possible to the function we want to integrate
  - intuition: the closer the pdf is to the integrand, the smaller in the ratios of f(x)/p(x)
  - most important variance reduction in rendering

$$\mathbf{x} \sim p(\mathbf{x}) \approx f(\mathbf{x})$$

- We will introduce later appropriate warp functions to sample according the advantageous pdfs

© 2020 Fabio Pellacini

Monte Carlo integration

- Pros: works for every domain
- Pros: convergence independent of dimensionality
- Pros: works for functions with discontinuities
- Pros: simple implementation

- Cons: relatively slow convergence
- Cons: variance (noise) in the estimate

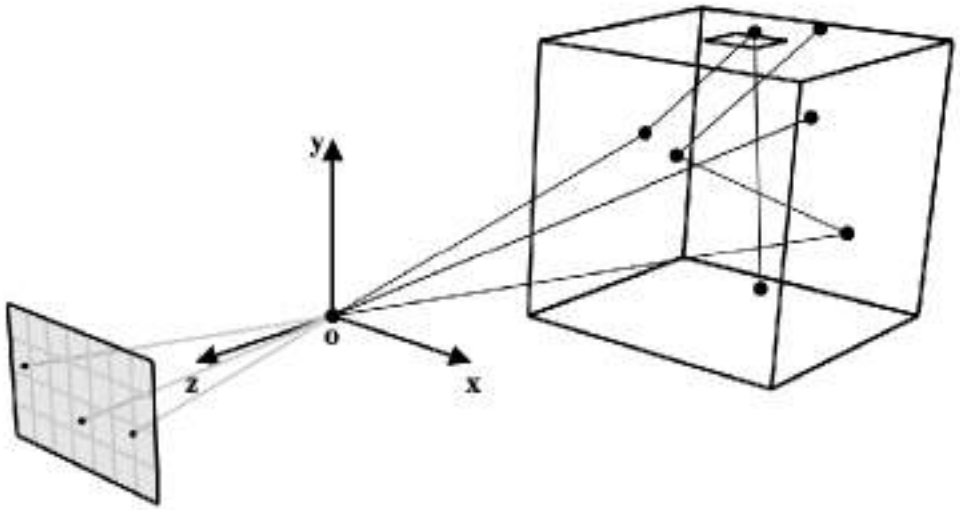
© 2020 Fabio Pellacini

Naive Path Tracing

© 2020 Fabio Pellacini

Path tracing

- Solves the rendering equation with Monte Carlo integration supporting all lighting effects shown above
- For each pixel independently, we solve the rendering equation by integrating the incoming radiance over several paths that go through that pixel – paths are traced backwards starting at the pixel and going towards the scene



© 2020 Fabio Pellacini

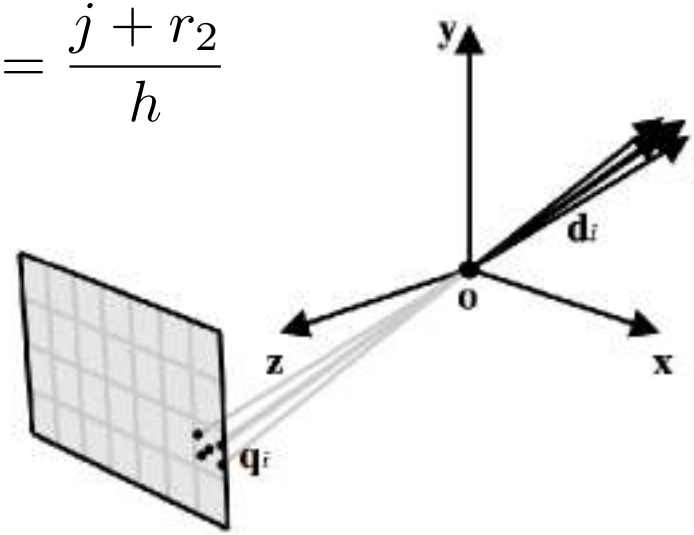
Pixel sampling

- We compute the pixel integral by straightforward Monte Carlo integration of the incoming radiance

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\mathbf{q} \in pixel} L_i(\mathbf{q}, \overrightarrow{\mathbf{eq}}) dA_{\mathbf{q}} \approx \frac{1}{N} \sum_{i}^{N} L_i(\mathbf{q}_i, \overrightarrow{\mathbf{eq}_i})$$

- To generate the random positions $\mathbf{q}_i$ we pick two random numbers in the unit interval and warp them to the camera *(u,v)* coordinates

$$u = \frac{i + r_1}{w} \quad v = \frac{j + r_2}{h}$$



© 2020 Fabio Pellacini

Estimating incoming radiance

- We estimate the incoming radiance by tracing a ray toward the scene
- If the ray hits, we return an estimate of the outgoing radiance computed at the intersection point and with the intersection BRDF
- If not, we return the background radiance at that direction

$$L_i(\mathbf{q}, \mathbf{d}) = \begin{cases} L(\mathbf{r}(\mathbf{q}, \mathbf{d}), -\mathbf{d}) & \text{if ray hits} \\ L_{environment}(\mathbf{d}) & \text{otherwise} \end{cases}$$

© 2020 Fabio Pellacini

Solving the reflection equation

- We first give an "intuitive" solution to the rendering equation that we will then formally prove later
- Consider the rendering equation

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- We can solve this equation by picking random direction in the sphere above the point – we use the hemisphere to handle opaque objects
- A Monte Carlo estimate for integral can be written as

$$L(\mathbf{x},\mathbf{o}) \approx L_e(\mathbf{x},\mathbf{o}) + \frac{L(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i) f(\mathbf{x},\mathbf{i}_i,\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i)}$$

© 2020 Fabio Pellacini

Hemispherical sampling

- To generate random directions in the hemisphere, we warp two uniform random numbers to an hemispherical direction
- To lower variance, we choose a distribution proportional to the cosine
- In local coordinates, we can write

$$\mathbf{i}^l = [\sqrt{1-r_2}\cos(2\pi r_1), \sqrt{1-r_2}\sin(2\pi r_1), r_2]$$

$$p(\mathbf{i}^l) = \frac{\mathbf{i}^l_z}{\pi}$$

- To get the world coordinates, we construct a frame with z aligned to the normal and transform by it

$$\mathbf{i} = \sqrt{1-r_2}\cos(2\pi r_1)\cdot\mathbf{t_x} + \sqrt{1-r_2}\sin(2\pi r_1)\cdot\mathbf{t_y} + r_2\cdot\mathbf{n}$$

$$p(\mathbf{i}) = \frac{\mathbf{n}\cdot\mathbf{i}}{\pi}$$

© 2020 Fabio Pellacini

Recursive evaluation

- The equation above cannot be directly evaluated since the unknown radiance is present on the right side
- To solve the equation, we recursively evaluate the integral using the same formulation
- The recursion terminates is we do not hit a surface, in which case we accumulate the radiance of the environment map
- But in most cases, say closed rooms, an infinite recursion will happen
  - for now, we stop evaluation after a fixed number of recursive calls
- We start the recursion at each camera pixel

© 2020 Fabio Pellacini

Path tracing – main loop

- We can now sketch the pseudocode for a naive path tracer starting from the outer loop

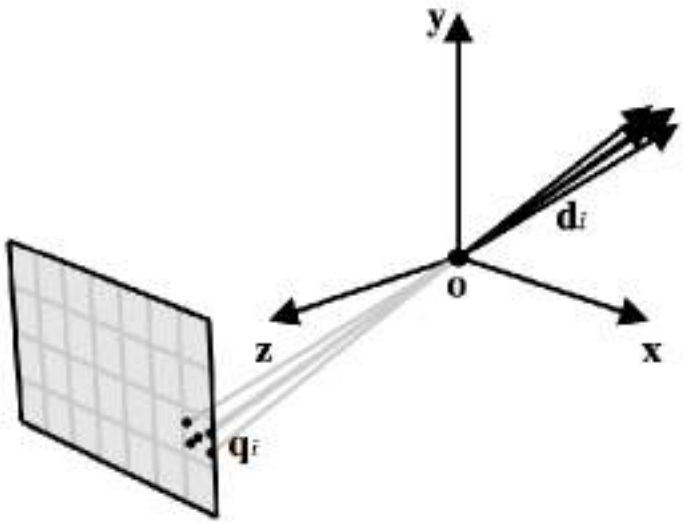
```
auto rngs = make_rngs();
auto acc = image{img.size, zero3f};
for(auto s : range(ns)) {
  for(auto j : range(img.height)) {
    for(auto i : range(img.width)) {
      auto puv = rand2f(rngs[i,j]);
      auto uv = (vec2f{i,j} + puv) / img.size;
      auto ray = eval_camera(cam, uv);
      acc[i,j] += shade(scene, ray, rngs[i,j]);
      img[i,j] = acc[i,j] / (s + 1);
    }
  }
  // view or save img if desired
}
```


© 2020 Fabio Pellacini

Path tracing – pixel sampling

- We then sample the evaluate the camera as before, ignoring the lens for now

```
ray3f eval_camera(camera* camera, vec2f image_uv) {
  auto q = vec3f{camera->film.x * (0.5 - image_uv.x),
    camera->film.y * (image_uv.y - 0.5), camera->lens};
  auto e = vec3f{0}; auto d = normalize(-q - e);
  return ray3f{transform_point(camera->frame, e),
      transform_direction(camera->frame, d)};
}
```



© 2020 Fabio Pellacini

Path tracing – environment maps

- If intersections fail, we can get the environment color by looking values in the environment texture using a lat-lon parametrization

$$L_{env}(\mathbf{i}^l) = k_e * txt\left[\frac{\text{atan2}(\mathbf{i}^l_z, \mathbf{i}^l_x)}{2\pi}, \frac{\text{acos}(\mathbf{i}^l_y)}{\pi}\right]$$

© 2020 Fabio Pellacini

Path tracing – environment maps

- If intersections fail, we can get the environment color by looking values in the environment texture using a lat-lon parametrization

```
vec3f eval_environment(environment* environment, vec3f dir) {
  auto local_dir = transform_direction(
                        inverse(environment->frame), dir)
  auto texcoord = vec2f{
        atan2(local_dir.z, local_dir.x) / (2 * pi),
        acos(clamp(local_dir.y, -1, 1)) / pi};
  if (texcoord.x < 0) texcoord.x += 1;
  return environment->emission *
            eval_texture(environment->emission_tex, texcoord);
}
```

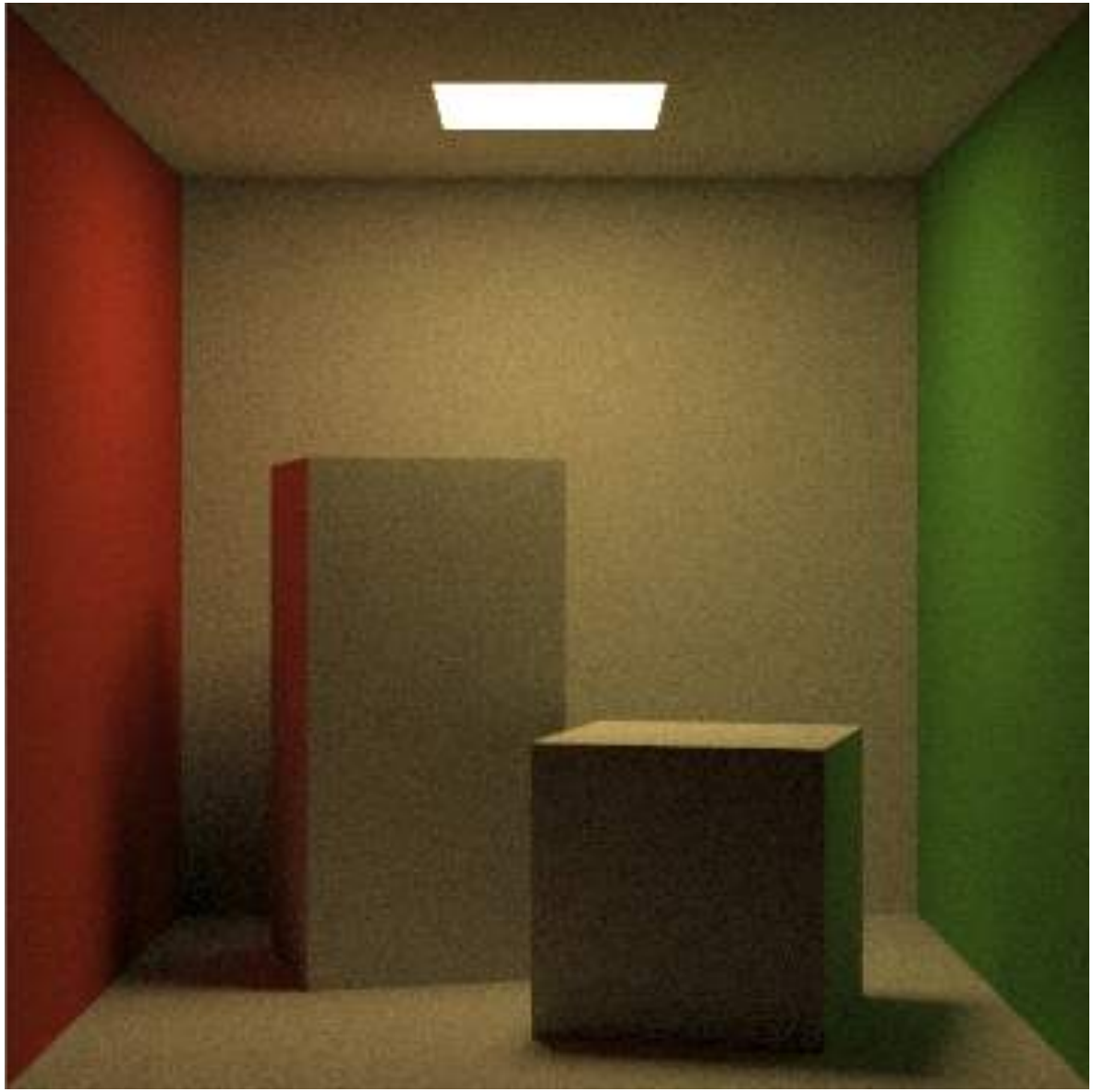

© 2020 Fabio Pellacini

Path tracing – recursive

- We can now write the path tracer with the previous equation

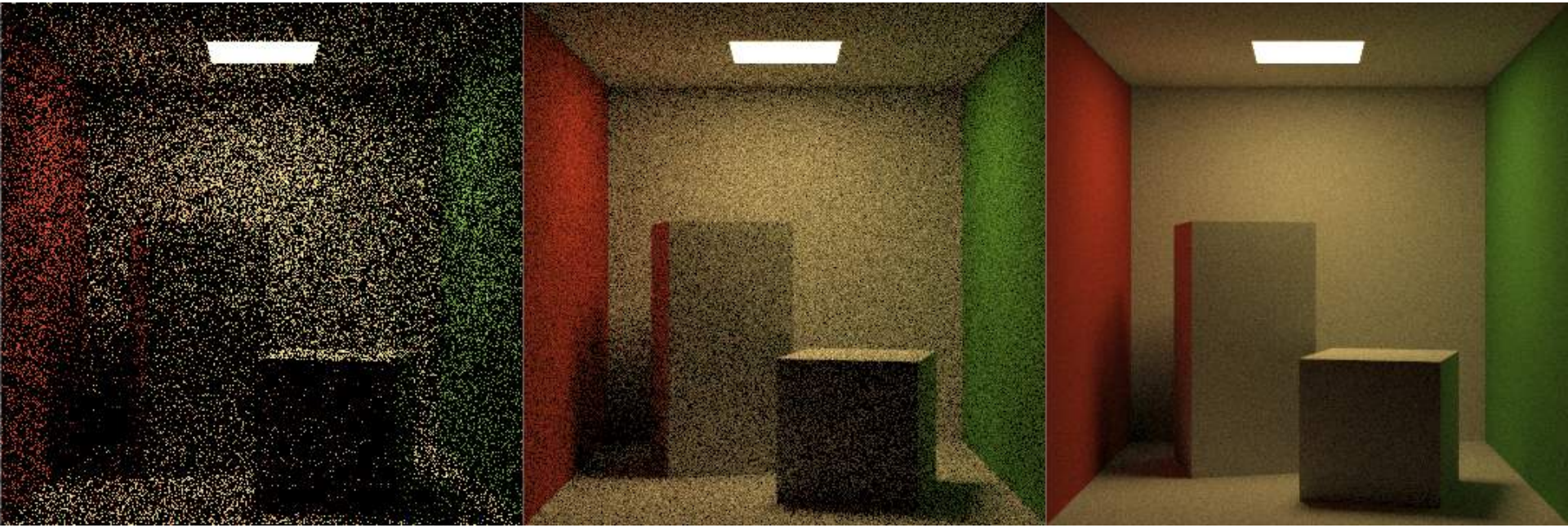
```
vec3f shade_recursive(scene* scn, ray3f ray, int bounce) {
  auto isec = intersect(scn, ray);
  if(!isec.hit) return eval_environment(scn, ray.d);

  auto [p, n] = eval_point(isec);    // eval pos, norm
  auto [e, f] = eval_material(isec); // eval bsdf
  auto o = -ray.d;

  auto l = eval_emission(e, n, o);   // for now, return le
  if(bounce >= max_bounce) return l; // exit

  auto i = sample_hemisphere_cos(n, rand2f(rng));
  l += shade_recursive(scn, {p, i, bounce+1}) *
       eval_brdf(f,n,i,o) * abs(dot(i,n)) /
       sample_hemisphere_cos_pdf(n,i);
  return l;
}
```


© 2020 Fabio Pellacini



© 2020 Fabio Pellacini

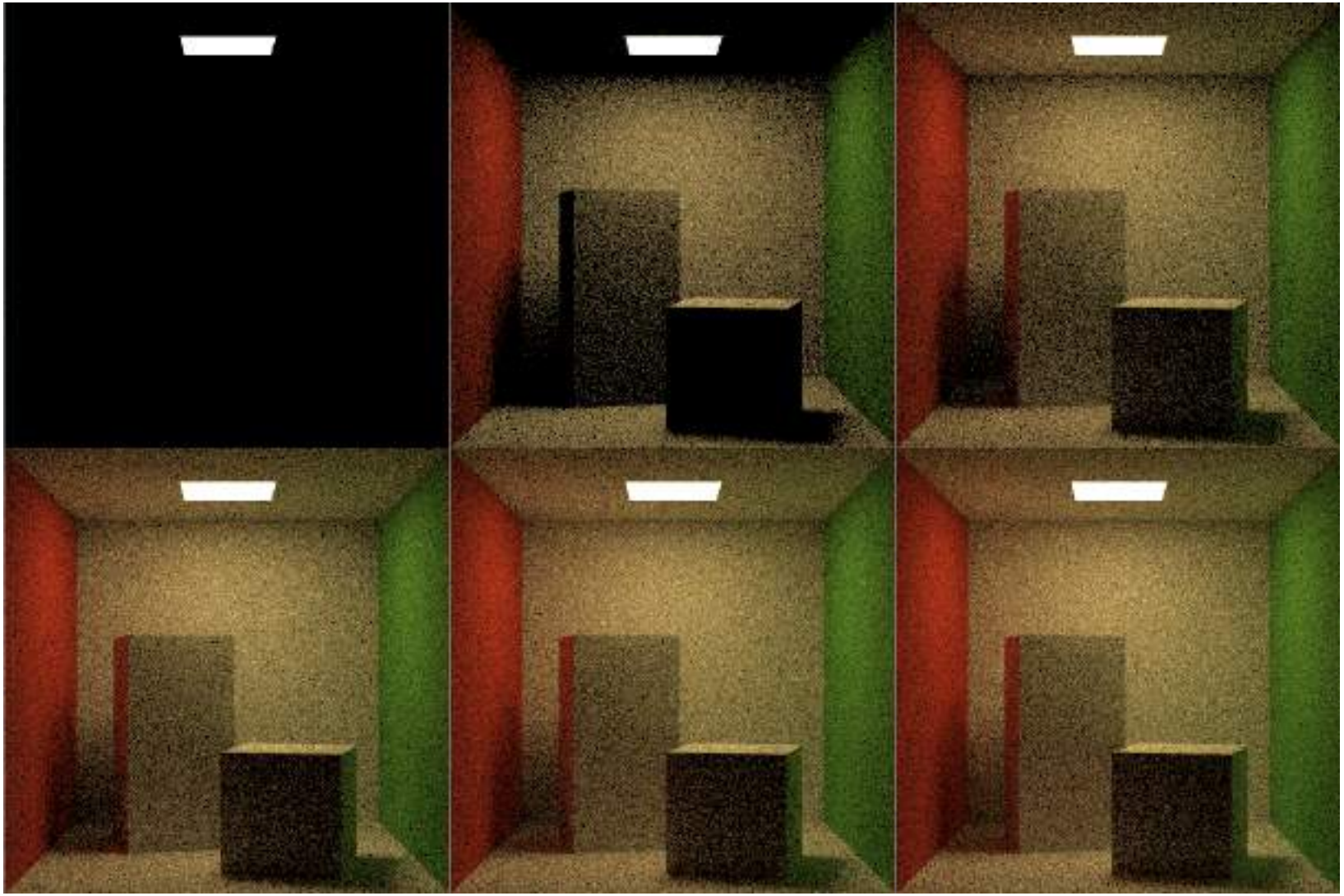
Naive path tracing – convergence



16 samples 256 samples 4096 samples

© 2020 Fabio Pellacini

Naive path tracing – bounces



© 2020 Fabio Pellacini

Russian roulette

- We want to avoid stopping at the arbitrary bounce since the energy associate with it depends on the scene configuration
- *Russian roulette*: stop with a given probability and scale the value of the remaining samples to the inverse of that probability to account for the energy loss due to stopping

$$g(\mathbf{x}) \text{ is an estimator for I}$$

$$g_r(\mathbf{x}) = \begin{cases} \frac{1}{p_r} g(\mathbf{x}) & \text{with probability } p_r \\ 0 & \text{with probability } 1 - p_r \end{cases}$$

$$E[g_r(\mathbf{x})] = p_r \cdot E[(1/p_r) g(\mathbf{x})] + (1 - p_r) \cdot 0 = E[g(\mathbf{x})]$$

© 2020 Fabio Pellacini

Russian roulette

- In the context of Path tracing, we check whether a path needs to be terminate and, if not, correct the recursive radiance
- For the probability of termination, we could use the surface reflectivity since that would match the real world behavior
- But computing that might not be easy – alternatively, we compute the survival probability as

$$p_r = \min\left(1, \frac{f(\mathbf{i}, \mathbf{o})|\mathbf{i} \cdot \mathbf{n}|}{p(\mathbf{i})}\right)$$

© 2020 Fabio Pellacini

Path tracing – Russian roulette

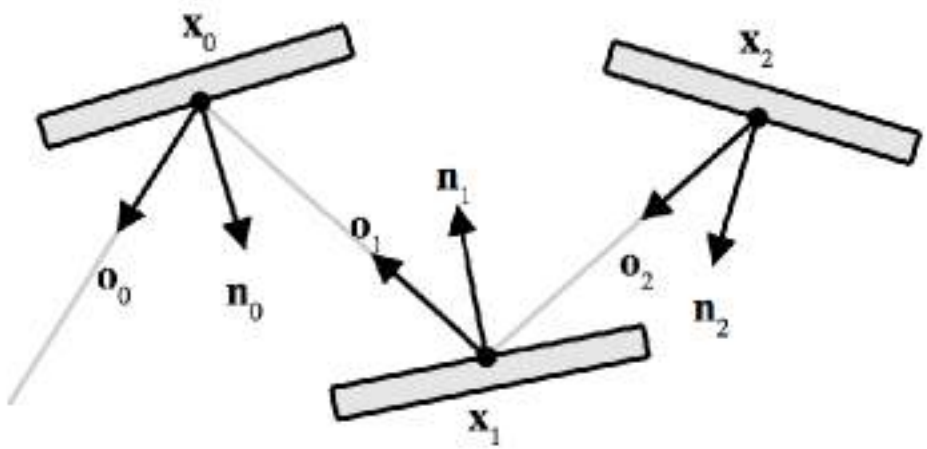
```
vec3f shade_russian_roulette(scene* scene, ray3f ray) {
  auto isec = intersect(scene, ray);
  if(!isec.hit) return eval_environment(scene, ray.d);

  auto [p, n]  = eval_point(isec);   // eval pos, norm
  auto [e, f] = eval_material(isec); // eval bsdf
  auto o = -ray.d;

  auto l = eval_emission(e, n, o);   // for now, return le
  if(bounce >= max_bounce) return l; // exit
  auto rr = min(1,max(f));           // continuation probability
  if(rand1f() >= rr) return l;       // russian roulette

  auto i = sample_hemisphere_cos(n, rand2f(rng));
  l += shade_recursive(scn, {x, i}) * eval_brdf(f,n,i,o) *
       abs(dot(i,n)) / (rr * sample_hemisphere_cos_pdf(n,i));
  return l;
}
```


© 2020 Fabio Pellacini

Path tracing – product form

- One problem of the recursive evaluation is that it is harder to improve since the recursion "hides" the overall path behavior
- An alternative way to write a path tracer is to unroll the recursion
- Let us write the Monte Carlo estimator for the *j*-th point along a path

$$L(\mathbf{x}_j, \mathbf{o}_j) = L_e(\mathbf{x}_j, \mathbf{o}_j) + [f(\mathbf{x}_j, \mathbf{i}_j, \mathbf{o}_j)(\mathbf{n}(\mathbf{x}_j) \cdot \mathbf{i}_j)/p(\mathbf{i}_j)]\, L(\mathbf{x}_{j+1}, \mathbf{o}_{j+1}) =$$
$$= L_e(\mathbf{x}_j, \mathbf{o}_j) + w_j L(\mathbf{x}_{j+1}, \mathbf{o}_{j+1}) \quad \text{with } \mathbf{i}_j = -\mathbf{o}_{j+1}$$



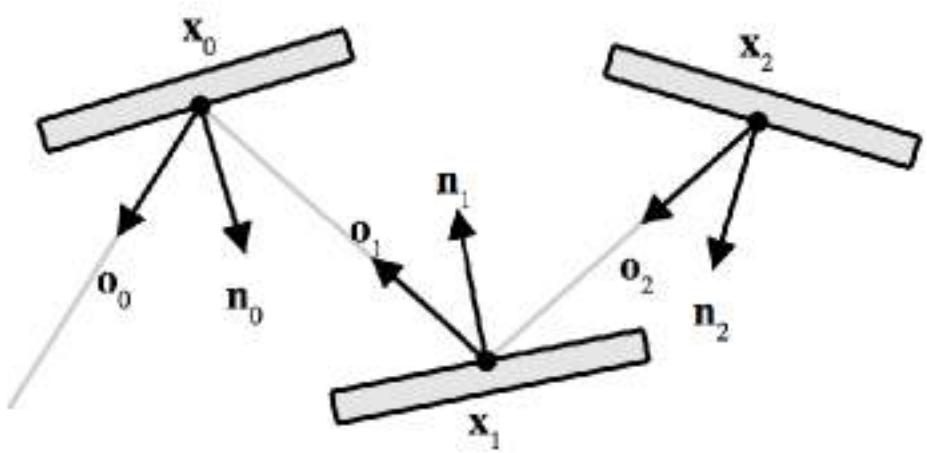
© 2020 Fabio Pellacini

Path tracing – product form

- To compute the radiance of the *whole path* we unroll the previous recursion as

$$L_{path} = L_e(\mathbf{x}_0, \mathbf{o}_0) + w_1 L_e(\mathbf{x}_1, \mathbf{o}_1) + w_1 w_2 L_e(\mathbf{x}_2, \mathbf{o}_2) + \ldots =$$

$$= \sum_j W_j L_e(\mathbf{x}_j, \mathbf{o}_j) \quad \text{with } W_j = w_j W_{j-1} = \prod_{k=0}^{j} w_k$$



© 2020 Fabio Pellacini

Path tracing - product form

```
vec3f shade_product(scene* scene, ray3f ray) {
  auto l = vec3f{0}, w = vec3f{1};
  for(auto bounce : range(max_bounce)) {
    auto isec = intersect(scene, ray);
    if(!isec.hit) {
      l += w * eval_environment(scene, ray.d); break; }
    auto [p, n] = eval_point(isec);    // eval pos, norm
    auto [e, f] = eval_material(isec); // eval bsdf
    auto o = -ray.d;                   // outgoing
    l += w * eval_emission(e, n, o);   // emission
    auto i = sample_hemisphere_cos(n, rand2f(rng)); // incoming
    w *= eval_brdf(f,n,i,o)  * abs(dot(i,n)) / // update weight
         sample_hemisphere_cos_pdf(n,i);
   if(rand1f() >= min(1,max(w))) break;  // russian roulette
    w *= 1 / min(1,max(w));              // update weight
    ray = {p, i};                        // recurse
 }
  return l;
}
```


© 2020 Fabio Pellacini

Path tracing - handle materials

- To handle sharp materials, we can add a special case to the rendering loop that samples their directions directly
- As a convenience, we can also fold the cosine inside the brdf functions
- Finally, we will later pick directions based on the full BRDF, not just the cosine

© 2020 Fabio Pellacini

```
vec3f shade_naive(scene* scene, ray3f ray) {
  auto l = zero3f, w = vec3f{1};
  for(auto bounce : range(max_bounce)) {
    auto isec = intersect(scene, ray);
    if(!isec.hit) {
      l += w * eval_environment(scene, ray.d); break; }
    auto [p, n]  = eval_point(isec);    // eval pos, norm
    auto [e, f] = eval_material(isec);  // eval bsdf
    auto o = -ray.d;                    // outgoing
    l += w * eval_emission(e, n, o);    // emission
    if(!is_delta(f)) { // sample smooth brdfs (fold cos into f)
      auto i = sample_brdfcos(f,n,o,rand2f(rng)); // incoming
      w *= eval_brdfcos(f,n,i,o) / sample_brdfcos_pdf(f,n,i,o);
    } else {          // sample sharp brdfs
      auto i = sample_delta(f,n,o,rand2f(rng));   // incoming
      w *= eval_delta(f,n,i,o) / sample_delta_pdf(n,i,o);
    }
    if(rand1f() >= min(1,max(w))) break;    // russian roulette
    w *= 1 / min(1,max(w));
    ray = {x, i};                           // recurse
 }
  return l;
}
```


© 2020 Fabio Pellacini

Path tracing – product form

```
vec3f shade_naive(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    <emission>
    if(!is_delta(f)) <handle smooth>
    else <handle delta>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


© 2020 Fabio Pellacini

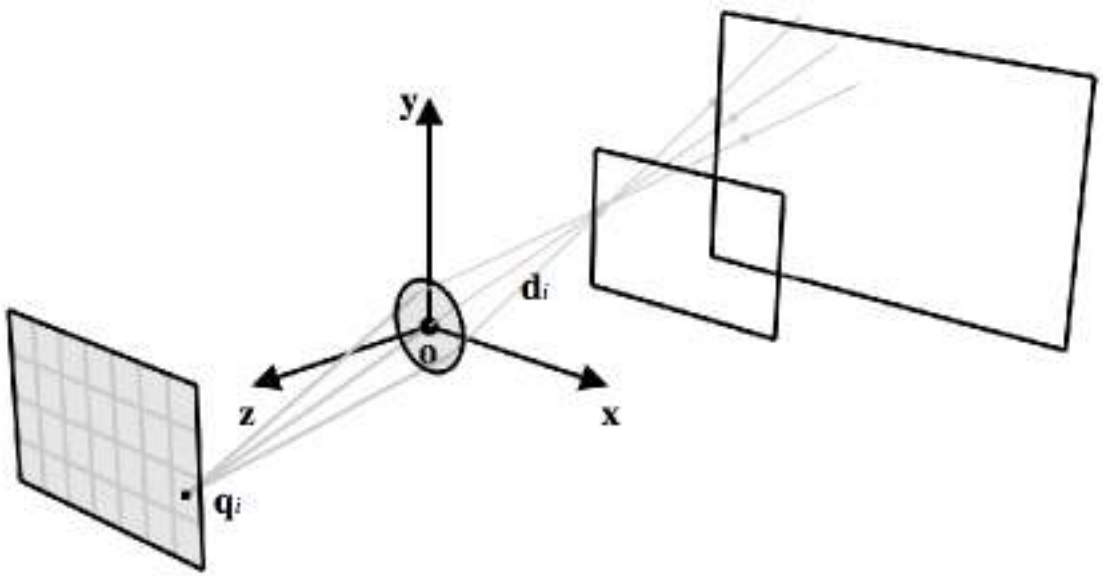
Realistic camera

© 2020 Fabio Pellacini

Pixel radiance

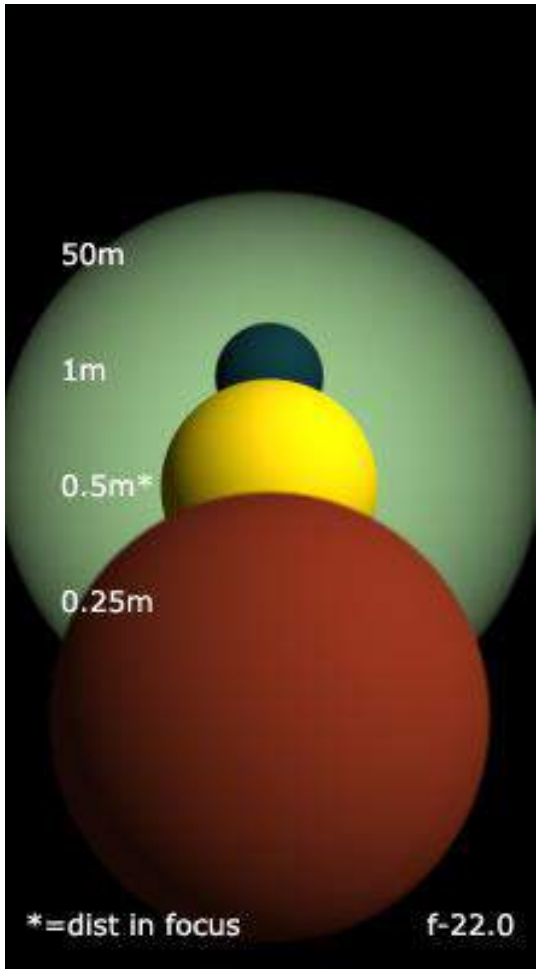
- We account for lenses with finite aperture by also integrating for all possible positions on the lens itself; this give us defocus blur

$$L_{pixel} = \frac{1}{A_{pixel}} \frac{1}{A_{lens}} \int_{\mathbf{q} \in pixel} \int_{\mathbf{e} \in lens} L_i(\mathbf{q}, \mathbf{d}) dA_{\mathbf{q}} dA_{\mathbf{e}} \text{ with } \mathbf{d} = \frac{\mathbf{e} - \mathbf{q}}{|\mathbf{e} - \mathbf{q}|}$$

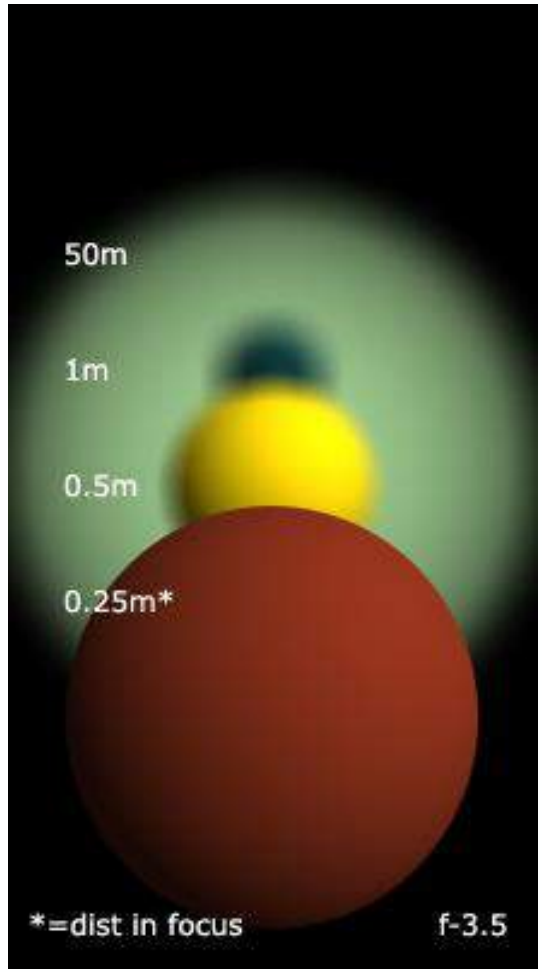


© 2020 Fabio Pellacini

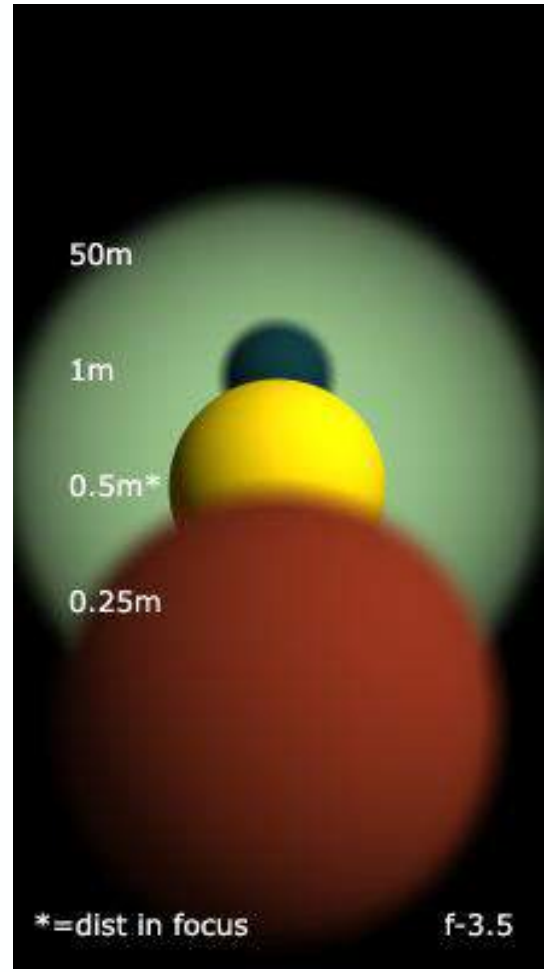
Pixel radiance



pinhole



close focus



far focus

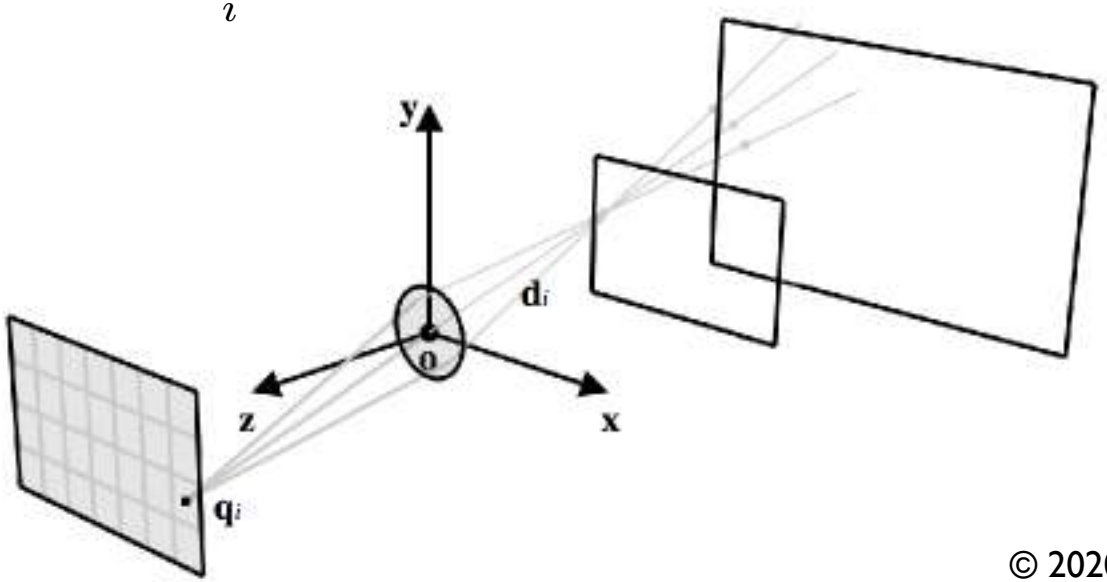
© 2020 Fabio Pellacini

Sampling the lens

- Monte Carlo integration works just fine when considering the lens too
  - note how we numerically compute two integrals at once

$$L_{pixel} = \frac{1}{A_{pixel}} \frac{1}{A_{lens}} \int_{\mathbf{q} \in pixel} \int_{\mathbf{e} \in lens} L_i(\mathbf{q}, \overrightarrow{\mathbf{eq}}) dA_{\mathbf{q}} dA_{\mathbf{e}}$$

$$\approx \frac{1}{N} \sum_{i}^{N} L_i(\mathbf{q}_i, \overrightarrow{\mathbf{e}_i \mathbf{q}_i})$$



© 2020 Fabio Pellacini

Sampling the lens

- For square lens, we could generate the lens positions by warping to a small square like we did not far – but lenses are really circular
- If we warp with polar coordinates we get points on the circles that are not equally distributed since the areas change
- To sample the circle, we *warp with an area-preserving method*

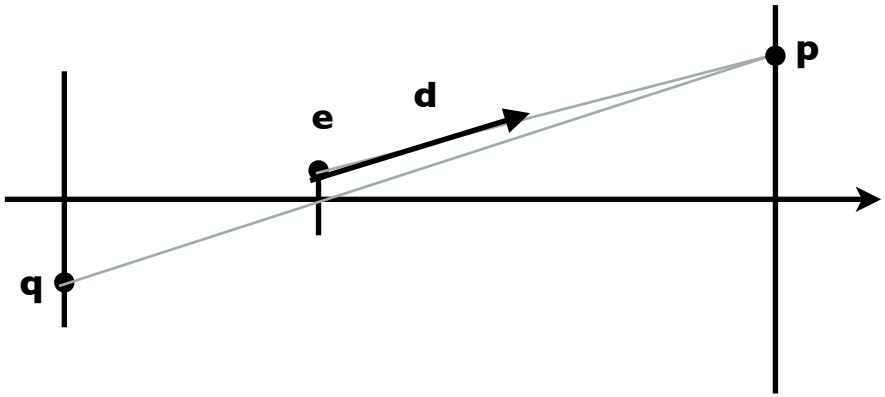
$$\mathbf{e}^l = [ar\cos\theta, ar\sin\theta, 0] \quad \text{with } \theta = 2\pi r_1, r = \sqrt{r_2}$$

© 2020 Fabio Pellacini

Sampling the lens

- To get the ray direction, we assume that we have a thin lens and connect the lens position to the position on the focal plane
  - pick a point on the film, compute direction as through lens center
  - find position on focus plane of the ray
  - pick a point on the lens, compute direction to focus point

$$\mathbf{d}^l = \frac{\overrightarrow{\mathbf{ep}}}{|\overrightarrow{\mathbf{ep}}|} \quad \mathbf{p}^l = \mathbf{c}^l \frac{f}{\mathbf{c}^l_z} \quad \mathbf{c}^l = -\frac{\mathbf{q}^l}{|\mathbf{q}^l|}$$



© 2020 Fabio Pellacini

Sampling the lens

```
ray3f eval_camera(camera* camera,vec2f image_uv,vec2f lens_uv) {
  // point on the image plane
  auto q = vec3f{camera->film.x * (0.5f - image_uv.x),
      camera->film.y * (image_uv.y - 0.5f), camera->lens};
  // ray direction through the lens center
  auto dc = -normalize(q);
  // point on the lens
  auto e = vec3f{lens_uv.x * camera->aperture / 2,
      lens_uv.y * camera->aperture / 2, 0};
  // point on the focus plane
  auto p = dc * camera->focus / abs(dc.z);
  // correct ray direction to account for camera focusing
  auto d = normalize(p - e);
  // done
  return ray3f{transform_point(camera->frame, e),
      transform_direction(camera->frame, d)};
}
```


© 2020 Fabio Pellacini

Sampling the lens

```
vec2f sample_disk(vec2f ruv) {
 return {cos(2 * pi * ruv.x) * sqrt(ruv.y),
         sin(2 * pi * ruv.x) * sqrt(ruv.y)};
}

ray3f sample_camera(camera* camera, vec2i ij, vec2i image_size,
                    vec2f puv, vec2f luv) {
  auto uv = vec2f{(ij.x + puv.x) / image_size.x,
                  (ij.y + puv.y) / image_size.y};
  return eval_camera(camera, uv, sample_disk(luv));
}

… main loop …
auto ray = sample_camera(camera, {i,j}, image.size,
                         rand2f(rng), rand2f(rng));
```


© 2020 Fabio Pellacini

Rendering animation

- In general the camera stays open for a finite amount of time while the objects may move in the scene; this gives us motion blur
- We can compute this effect by integrating over time and considering a time-varying scene, say due to animation

$$L_{pixel} = \frac{1}{A_{pixel}} \frac{1}{A_{lens}} \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} \int_{\mathbf{q} \in pixel} \int_{\mathbf{e} \in lens} L_i(\mathbf{q}, \mathbf{d}, t) dA_{\mathbf{q}} dA_{\mathbf{e}} dt$$

© 2020 Fabio Pellacini

Direct Illumination

© 2020 Fabio Pellacini

Splitting direct and indirect

- The main concern of the previous renderer is that all paths that do not touch the light will end up with zero contribution
- Solution: connect each point in the path to at least one light
  - in jargon this is called next event estimation
- To do this, we split the integral of the reflected radiance *direct illumination*, i.e. the reflected radiance originating directly from light sources, and *indirect illumination*, i.e. all the rest
- For indirect illumination, we will proceed as before
- For direct illumination, we want to sample the lights directly to ensure that we hit one

© 2020 Fabio Pellacini

Splitting direct and indirect

- Let us now write the reflection equation by splitting direct and indirect illumination

$$
\begin{aligned}
L_r(\mathbf{x},\mathbf{o}) &= \int_{H^2} L_i(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} = \\
&= \int_{H^2} L_e(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} + \\
&+ \int_{H^2} L_r(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} = \\
&= L_d(\mathbf{x},\mathbf{o}) + L_{ind}(\mathbf{x},\mathbf{o})
\end{aligned}
$$

- Direct illumination can be thought of as the solution of the reflection equation where the incoming illumination is just the emission term

$$
L_d(\mathbf{x},\mathbf{o}) = \int_{H^2_\mathbf{x}} L_e(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i}
$$

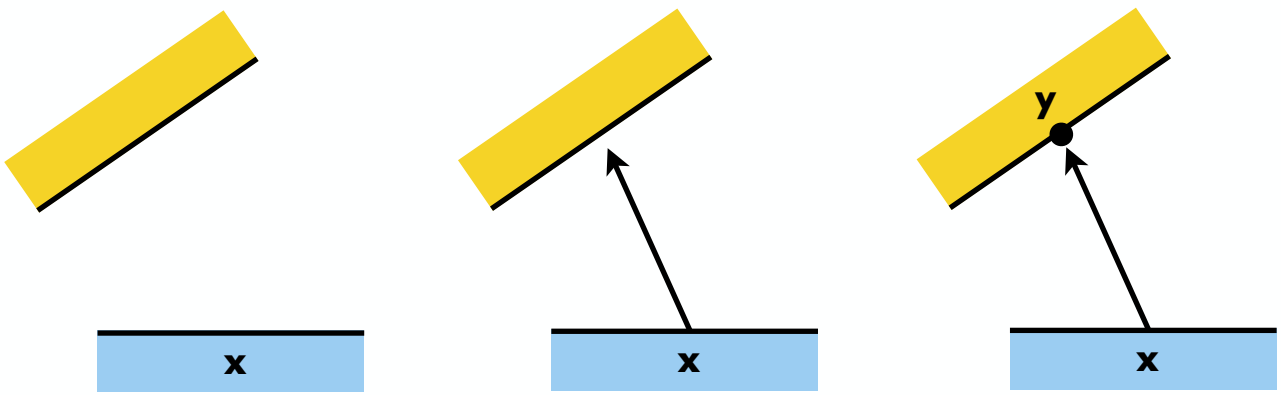
© 2020 Fabio Pellacini

Direct illumination – hemisphere

- We can solve the direct illumination equation with Monte Carlo by picking random directions and evaluating the integrand function

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i)}$$

- The problem here is becomes obvious – each ray with L_e zero has no contribution



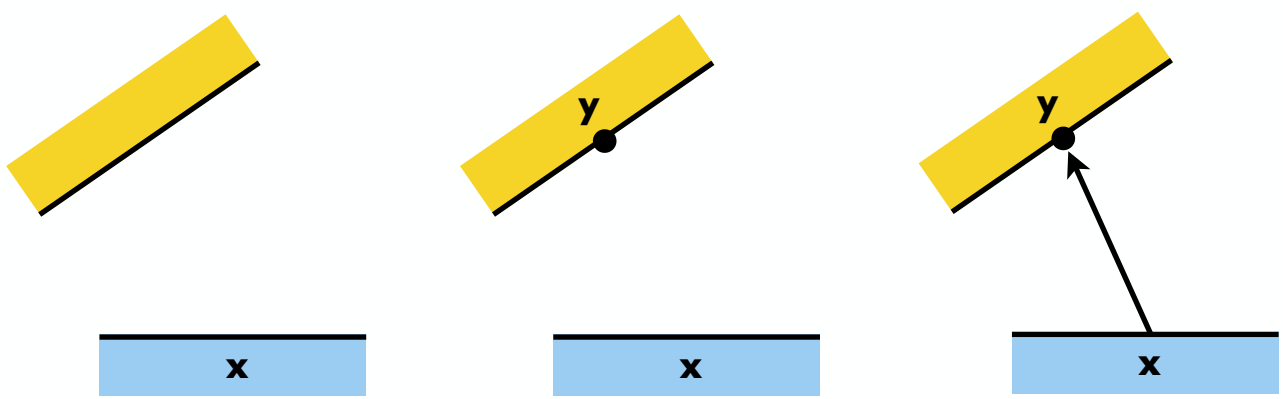
© 2020 Fabio Pellacini

Direct illumination – light

- A better alternative is to pick a point on the light and compute the direction from that;

$$L_d(\mathbf{x},\mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i)f(\mathbf{x},\mathbf{i_i},\mathbf{o})|\mathbf{n_x}\cdot\mathbf{i}_i|}{p(\mathbf{i}_i|\mathbf{y}_i)} \qquad \mathbf{i}_i = \frac{\mathbf{y}_i-\mathbf{x}}{|\mathbf{y}_i-\mathbf{x}|}$$

- We still need to cast a ray to check whether we did hit the light
- We now have to compute the direction PDF from the point PDF



© 2020 Fabio Pellacini

Sampling the light surface

- To evaluate it, we have to pick points uniformly over the light surface
- For a square light of area *A*, we can warp two random numbers as

$$\mathbf{y}^l = \sqrt{A}[r_1 - 0.5, r_2 - 0.5, 0]$$
$$\mathbf{y}^l \sim p(\mathbf{y}^l) = \frac{1}{A}$$

- To move to world coordinates, we use a frame that describes the orientation of the square light

$$\mathbf{y} = \sqrt{A}(r_1 - 0.5) \cdot \mathbf{F_x} + \sqrt{A}(r_2 - 0.5) \cdot \mathbf{F_y} + \mathbf{F_o}$$
$$\mathbf{y} \sim p(\mathbf{y}) = \frac{1}{A}$$

© 2020 Fabio Pellacini

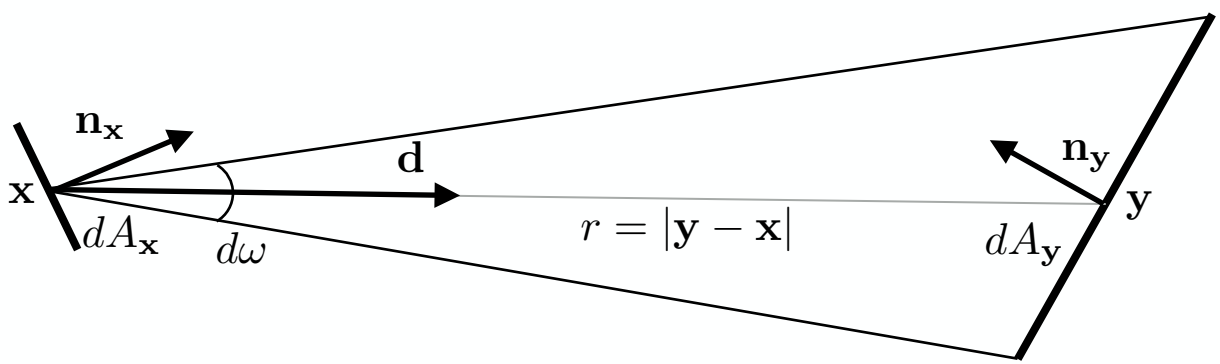
Sampling the light surface

- To compute the PDF of picking a direction from a point, we consider the differential solid angle subtended by a surface w.r.t. to another

$$d\omega_{\mathbf{i}} = \frac{dA_{\mathbf{y}}|-\mathbf{i}\cdot\mathbf{n}_{\mathbf{y}}|}{|\mathbf{y}-\mathbf{x}|^2}$$

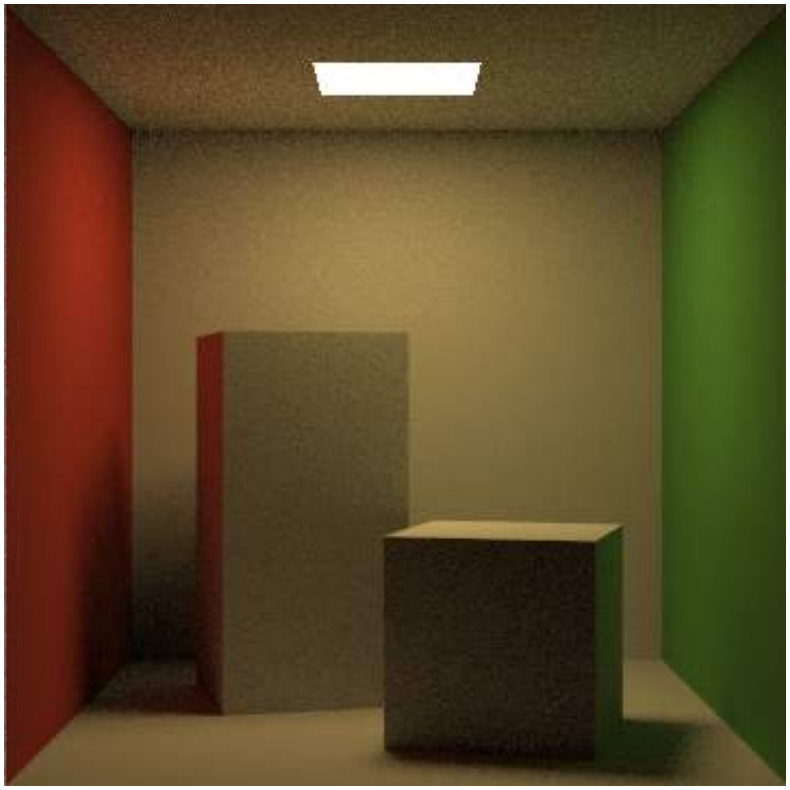
- From this, we derive the PDF of the direction as the product of the PDF of picking the point and the change of variable above

$$p(\mathbf{i}|\mathbf{y}) = p(\mathbf{y})\frac{|\mathbf{y}-\mathbf{x}|^2}{|-\mathbf{i}\cdot\mathbf{n}_{\mathbf{y}}|} = \frac{|\mathbf{y}-\mathbf{x}|^2}{A|-\mathbf{i}\cdot\mathbf{n}_{\mathbf{y}}|}$$

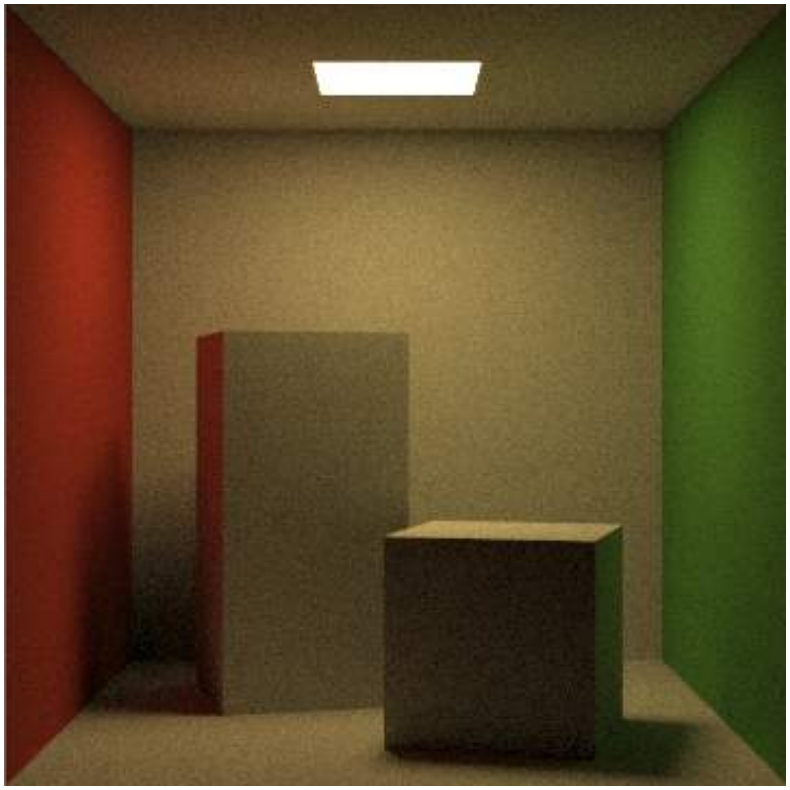


© 2020 Fabio Pellacini

Hemispherical vs light



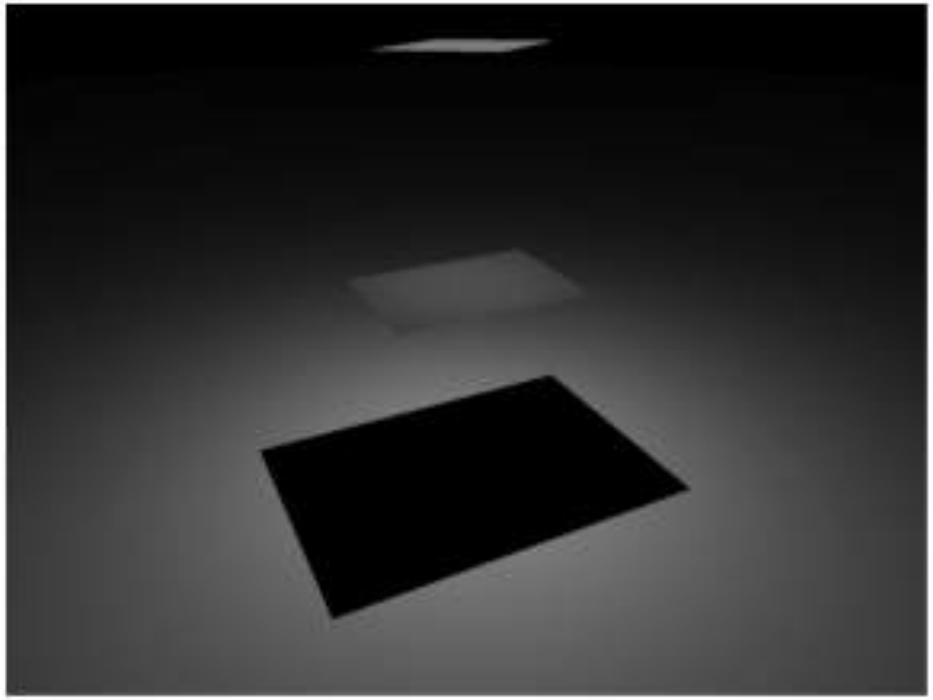
64 light samples



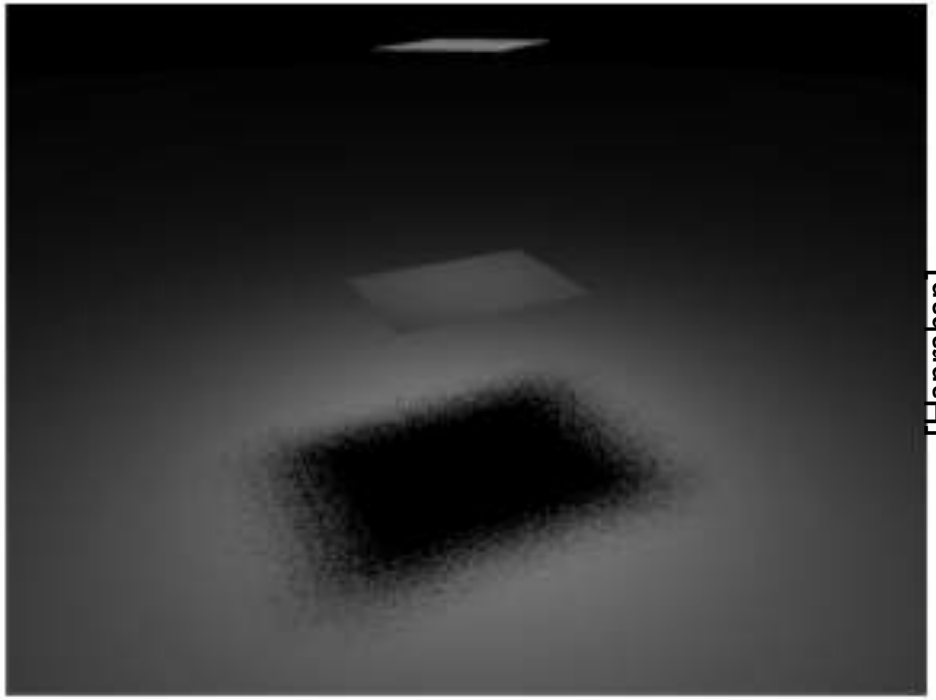
4096 hemispherical samples

© 2020 Fabio Pellacini

Monte Carlo vs quadrature



1 sample – center

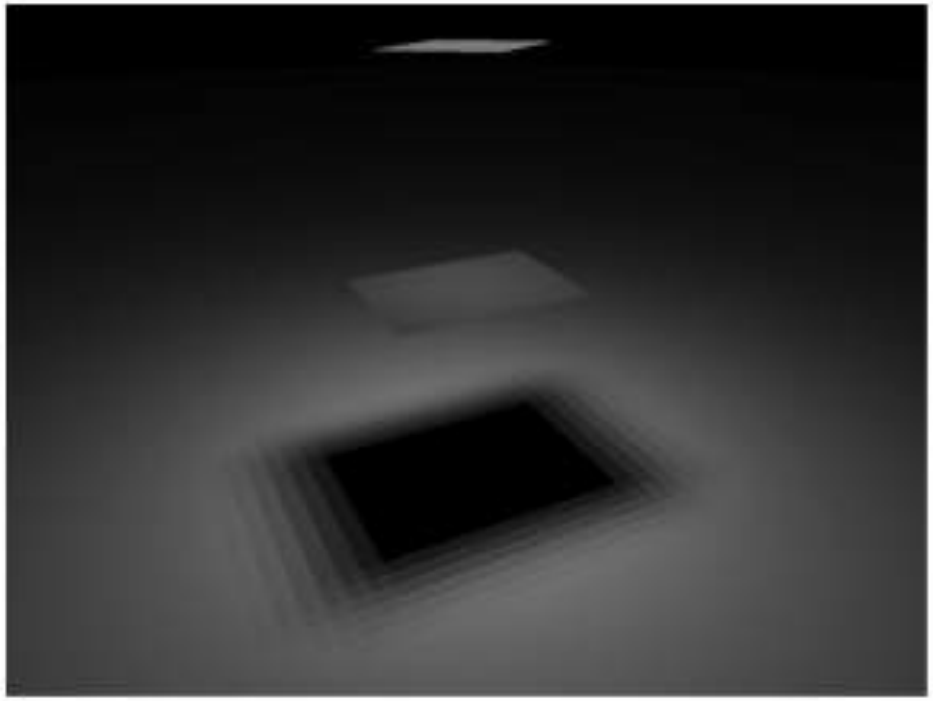


16 samples – random

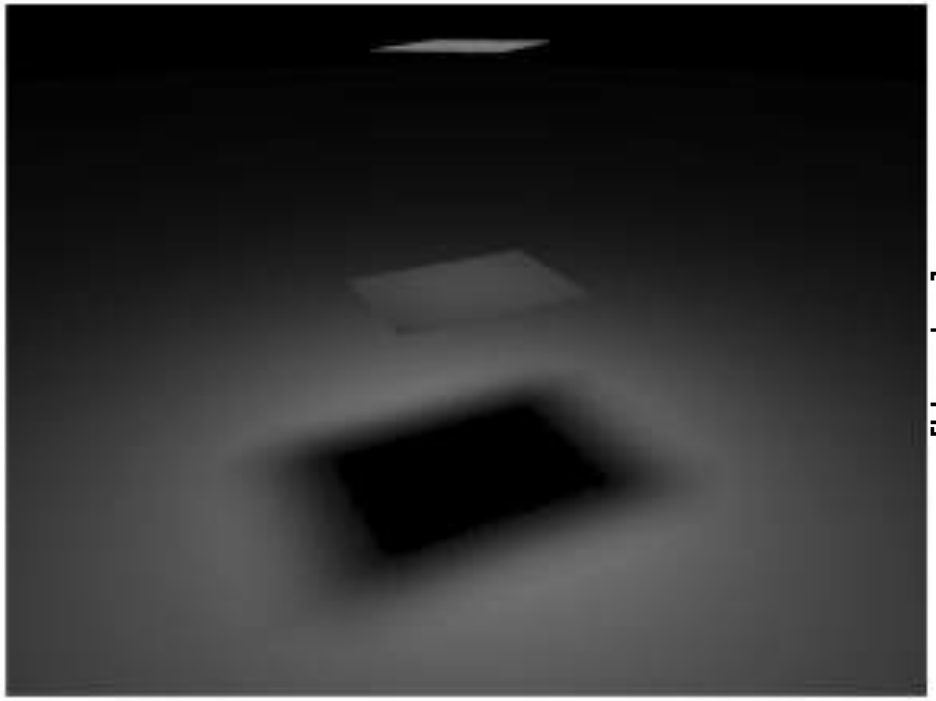
[Hanrahan]

© 2020 Fabio Pellacini

Monte Carlo vs quadrature



16 samples – center



16 samples – stratified

[Hanrahan]

© 2020 Fabio Pellacini

Sampling multiple lights

- To handle multiple lights, we can estimate their contributions independently since each light integral is independent

$$L_d(\mathbf{x},\mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_e(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}| d\omega_{\mathbf{i}} =$$

$$= \sum_l^{n_l} \int_{H^2_{\mathbf{x}}} L_e^l(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- We can estimate this by picking a direction for each light independently

$$L_d(\mathbf{x},\mathbf{o}) \approx \sum_l^{n_l} \frac{L_e^l(\mathbf{r}(\mathbf{x},\mathbf{i}_i^l),-\mathbf{i}_i^l) f(\mathbf{x},\mathbf{i}_i^l,\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}_i^l|}{p(\mathbf{y}_i^l|\mathbf{i}_i^l)}$$

© 2020 Fabio Pellacini

Sampling multiple lights

- As the number of lights increases, this formulation becomes inefficient
- The alternative is to pick a *light at random for each sample*
  - here we are sampling from a uniform *discrete* distribution

$$l_i = \text{floor}(r \cdot n_l) \quad l \sim p(l) = \frac{1}{n_l}$$

- We can estimate the illumination by considering our light choice and folding its PDF into the complete PDF

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e^{l_i}(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \mathbf{y}_i | l_i)}$$

$$p(\mathbf{i}|\mathbf{y}|l) = p(\mathbf{i}|\mathbf{y})p(l) = \frac{|\mathbf{y} - \mathbf{x}|^2}{n_l A_l | - \mathbf{i} \cdot \mathbf{n}_\mathbf{y}|}$$

© 2020 Fabio Pellacini

Path tracing – direct illumination

- Integrate direct illumination in the path tracer by just inserting it in the main loop; emission is now computed only only first bounce since for other bounces it is integrated into the direct illumination estimation

```
vec3f shade_direct(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    if(!bounce) <emission>    // direct only on first bounce
    if(bounce < max_bounce-1) <handle direct>
    if(!is_delta(f)) <handle smooth>
    else <handle delta>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


© 2020 Fabio Pellacini

Path tracing – direct illumination

- Here we assume lights are only quads and that we have stored a list of objects that have emissive materials.

```
<handle direct> {
  // pick a light, a quad index and a quad uv
  auto light = sample_uniform(scene->lights, rand1f(rng));
  auto qid = sample_uniform(light->quads.size(), rand1f(rng));
  auto uv  = rand2f(rng);
  // evaluate the light point and material
  auto [lp, ln] = eval_point(light, qid, uv);
  auto [le, _ ] = eval_material(light, qid, uv);
  auto li = normalize(lp - p); auto ld = length(lp - p);
  // if the light is visible, accumulate its contribution
  if(!intersect(scene, {p, li, eps, ld})) {
  auto lpdf = (ld*ld) / (dot(li,nl) * quad_area(light,quad) *
      scene->lights.size() * light->quads.size());
    l += w * eval_emission(le, ln, -li) *
            eval_brdfcos(f, n, li, o) * dot(n, li) / lpdf;
 }
}
```

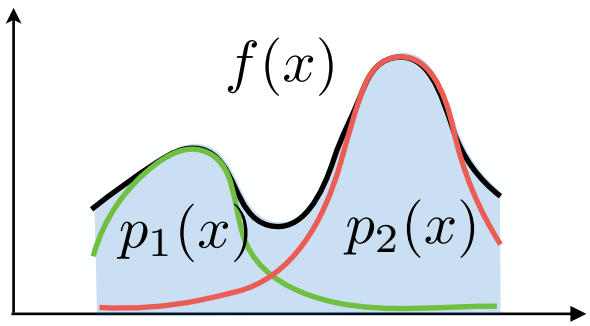

© 2020 Fabio Pellacini

Multiple Importance Sampling

© 2020 Fabio Pellacini

Multiple importance sampling

- We want to sample a complex function with importance sampling, but we do not know hoe to sample it well over the entire domain
- But we can sample different parts of the domain well with different sampling techniques
- We want to combine all techniques into a single robust one
- *Multiple importance sampling*: pick a technique at random and sample according to that that one
- Insight: avoid issues when a technique would not sample well a region



© 2020 Fabio Pellacini

Multiple importance sampling

- Simple formulation: pick a technique at random, based on predetermined probabilities, and sample according to that that one
- Consider *M* sampling techniques, each with pdf p_j, that we want to sample with probability w_j
- Pick a distribution at random with probability w_j
- Pick a sample from that distribution
- Compute the combined pdf as weighted sum of all pdfs

$$E[I] = \frac{1}{N} \sum_i^N \frac{f(\mathbf{x}_i)}{\sum_j^M w_j p_j(\mathbf{x}_i)}$$

$$j \sim \{w_j\} \rightarrow \mathbf{x}_i \sim p_j(\mathbf{x})$$

$$\sum_j w_j = 1$$

© 2020 Fabio Pellacini

Multiple importance sampling

- General formulation: pick samples from multiple techniques
- Consider *M* sampling techniques with pdfs p_j
- Pick N_j samples from each technique
- The combined estimator for these samples is

$$E[I] = \sum_j^M \frac{1}{N_j} \sum_i^{N_j} w_j(\mathbf{x}_{j,i}) \frac{f(\mathbf{x}_{j,i})}{p_j(\mathbf{x}_{j,i})}$$

$$\mathbf{x}_{j,i} \sim p_j(\mathbf{x})$$

$$\sum_j w_j(\mathbf{x}) = 1$$

- Choose weights w_j to reduce variance by applying heuristics
- Balance heuristic: near optimal weights

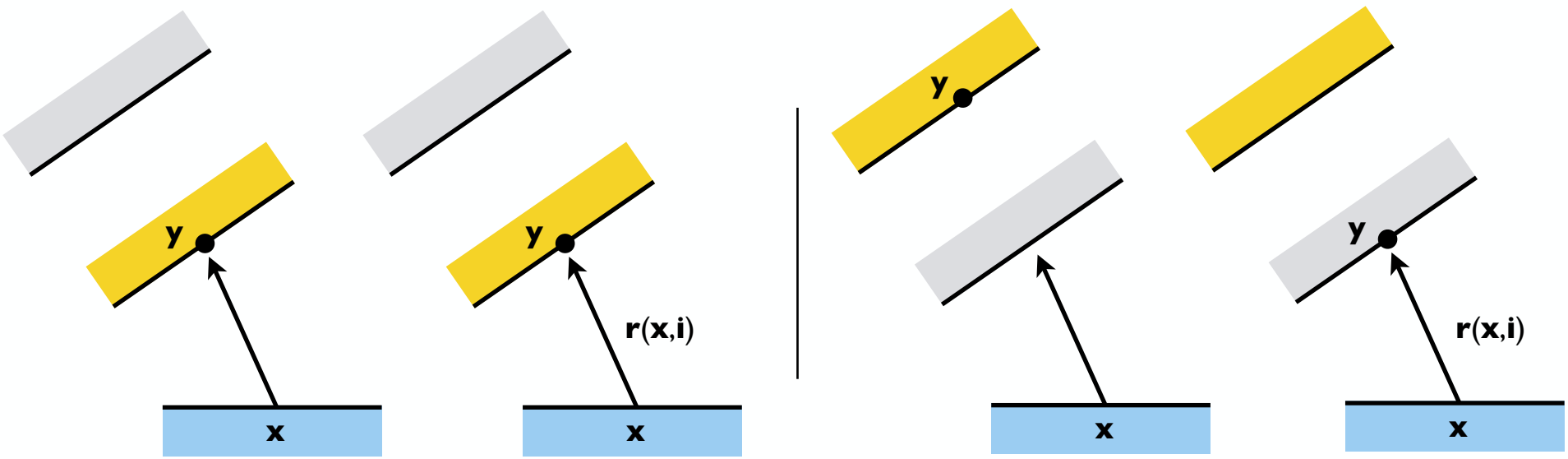
$$w_j(\mathbf{x}) = \frac{N_j p_j(\mathbf{x})}{\sum_k^M N_k p_k(\mathbf{x})}$$

© 2020 Fabio Pellacini

Sampling multiple lights – MIS

- A drawback of the previous formulation is that we might waste samples for lights that are occluded by other lights since we consider illumination only from the light we have chosen

$$L_d(\mathbf{x},\mathbf{o}) \approx \frac{L_e^{l_i}(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i)f(\mathbf{x},\mathbf{i}_i,\mathbf{o})|\mathbf{n}_\mathbf{x}\cdot\mathbf{i}_i|}{p(\mathbf{i}_i|\mathbf{y}_i|l_i)}$$

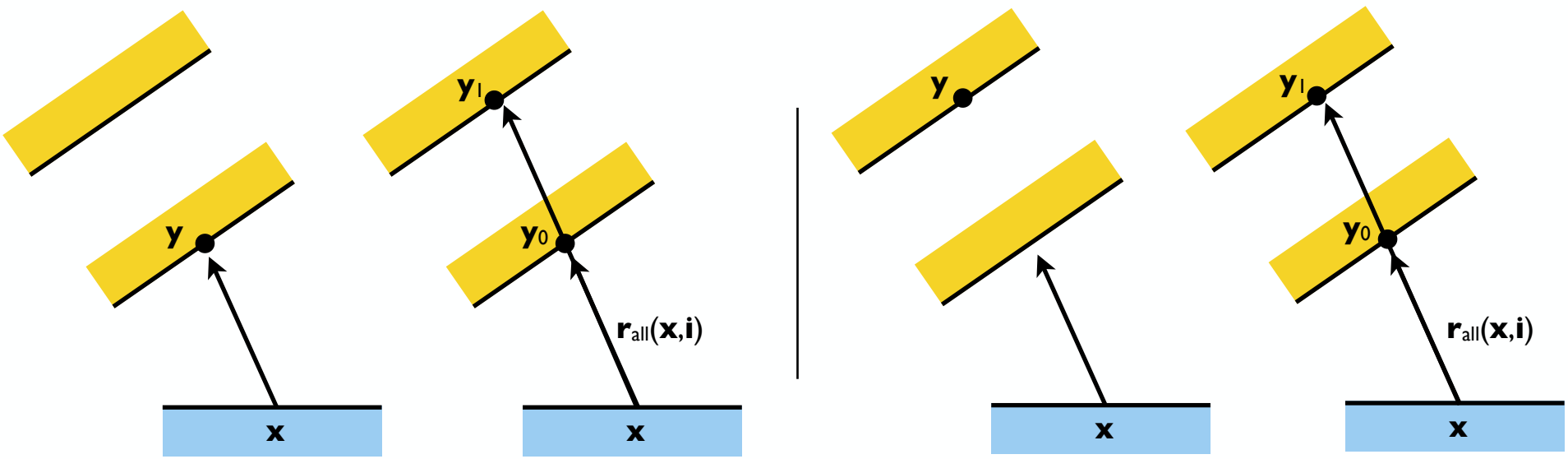


© 2020 Fabio Pellacini

Sampling multiple lights – MIS

- The alternative is to always consider the first visible light, but correct the PDF to account for the fact that the sampled direction *i* might have come from any point y_j along the ray from the shading point *x*

$$L_d(\mathbf{x},\mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i)f(\mathbf{x},\mathbf{i}_i,\mathbf{o})|\mathbf{n}_\mathbf{x}\cdot\mathbf{i}_i|}{p(\mathbf{i}_i|\{\mathbf{y}_l\})} \qquad \{\mathbf{y}_l\} = \mathbf{r}_{\text{all}}(\mathbf{x},\mathbf{i}_i)$$



© 2020 Fabio Pellacini

Sampling multiple lights – MIS

- This is an example of multiple importance sampling, where we pick a direction based on different sampling strategies, in this case we have one distribution for each light

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \{\mathbf{y}_l\})} \qquad \{\mathbf{y}_l\} = \mathbf{r}_{\mathrm{all}}(\mathbf{x}, \mathbf{i}_i)$$

- We can compute the PDF by summing the PDFs for each light weighted by the probability of picking that light

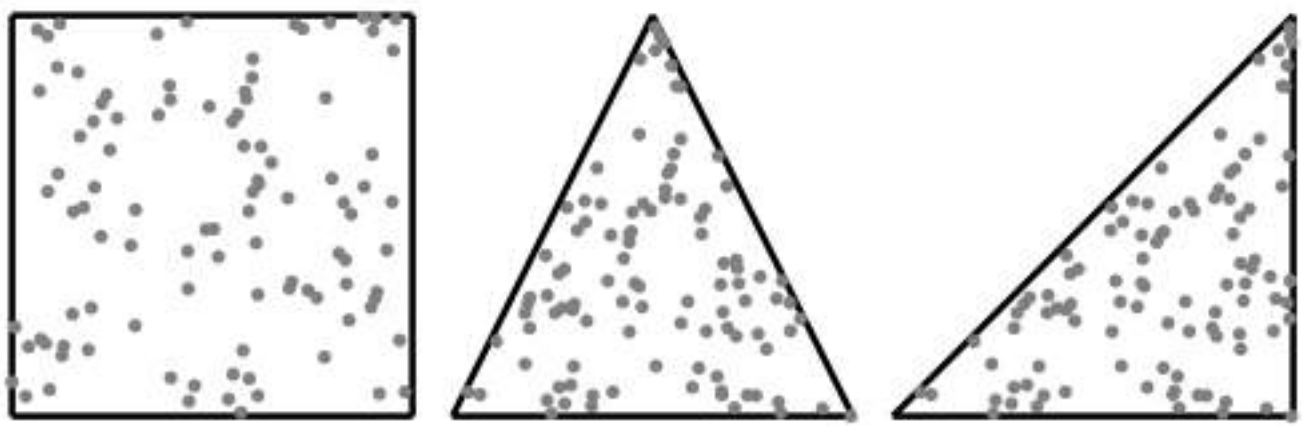
$$p(\mathbf{i}, \{\mathbf{y}_l\}) = \sum_{\mathbf{y}_l \in \mathbf{r}_{\mathrm{all}}(\mathbf{x}, \mathbf{i})} p(\mathbf{i} | \mathbf{y}_l) p(l) = \frac{|\mathbf{y}_l - \mathbf{x}|}{n_l A_l | - \mathbf{i} \cdot \mathbf{n}_{\mathbf{y}_l}|}$$

© 2020 Fabio Pellacini

Sampling triangle meshes

- In general, our surfaces are described by triangle meshes
- We want to a procedure to pick point *uniformly* on the entire surface
- Let us start by picking points on a triangle with uniform probability by warping from the unit square

$$\mathbf{y} = \sqrt{r_0}(1 - r_1)\mathbf{v}_0 + (1 - \sqrt{r_1})\mathbf{v}_1 + r_1\sqrt{r_0}\mathbf{v}_2$$



© 2020 Fabio Pellacini

Sampling triangle meshes

- Now we want to pick a triangle at random from the set of triangles with probability proportional to its area
- This is a non-uniform discrete distribution
- To sample it, we warp a random number in [0,1) to the distribution by
  - compute the triangle areas
  - from these, derive the cdf of the distribution
  - for each sample, we use a random number in [0,1) to find the smallest cdf value that is larger than the given one
  - its index is the wanted triangle index

$$p_j = \frac{A_j}{\sum_k A_k} \quad c_j = \sum_{k=0}^{j} p_k \quad t_i = \arg\min_j c_j \geq r$$

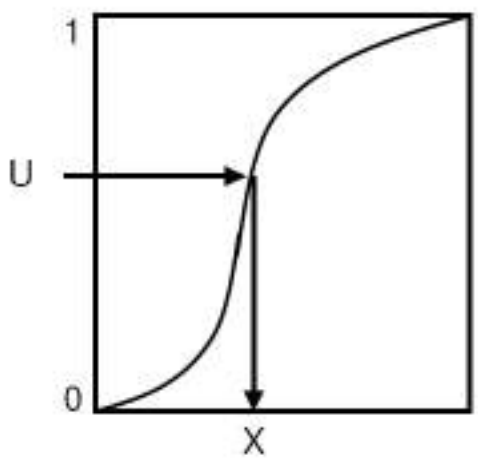
© 2020 Fabio Pellacini

Sampling discreet distribution

- Theorem: given the uniform random variable *U* in [0,1) and a cumulative distribution function *c*, then the random variable $X=c^{-1}(U)$ has *c* as its distribution

$$U \in [0,1) \sim p_U = 1 \rightarrow X = c^{-1}(U) \sim p_X = c$$

- To sample a random variable according to a distribution *p* do
  - compute its cumulative distribution function *c* and its inverse c^{-1}
  - use c^{-1} to warp uniform samples to the new domain

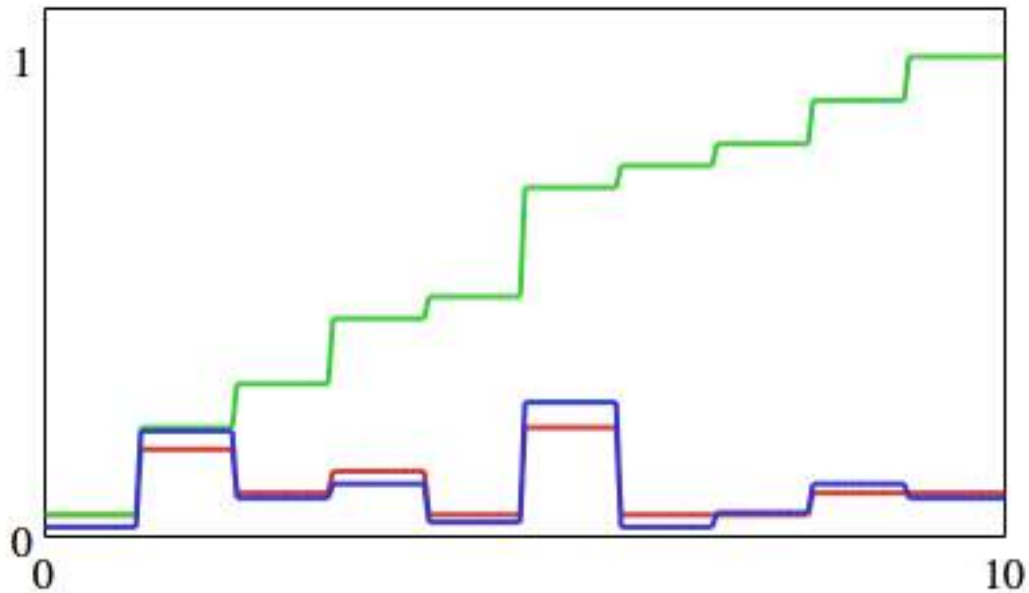


[Krivanec]

© 2020 Fabio Pellacini

Sampling triangle meshes

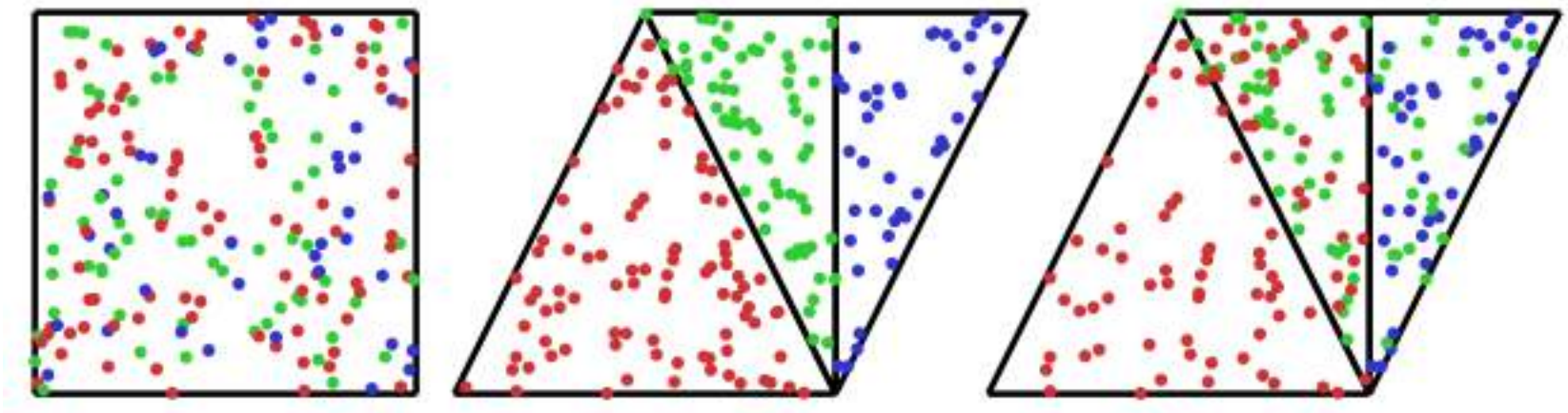
- Example case with 10 triangles



© 2020 Fabio Pellacini

Sampling triangle meshes

- Comparison between sampling uniformly the triangle index or sampling proportionally to area



© 2020 Fabio Pellacini

Environment map

- So far we sampled environment maps uniformly from the surface
- But, just like small lights, small areas of high values in the environment will likely be missed when sampling
- As an alternative, we can consider the environment as a light and compute directions by picking its pixels with probability proportional to the pixel intensity weighted by its size
- This is a discreet distribution similar to the one used to pick between triangles based on their areas, so we can use the same procedure

$$p_{(i,j)} = |\text{texture}(i,j)| \sin(\pi j/h)$$

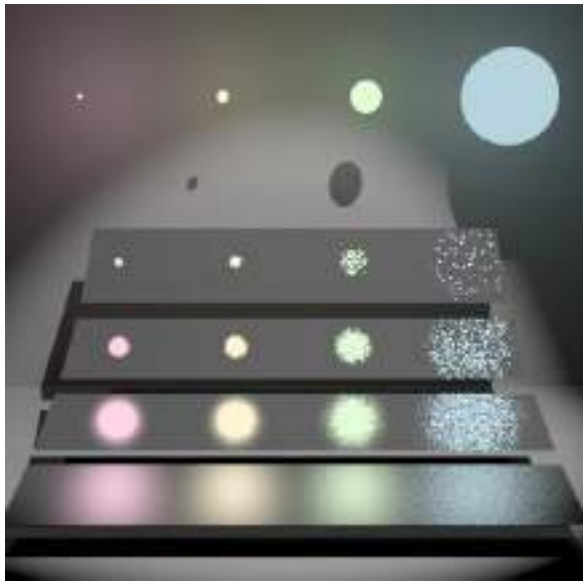
$$\mathbf{i}^l = [\cos(2\pi i/w)\sin(\pi j/h), \cos(\pi j/h), \sin(2\pi i/w)\sin(\pi j/h)]$$

© 2020 Fabio Pellacini

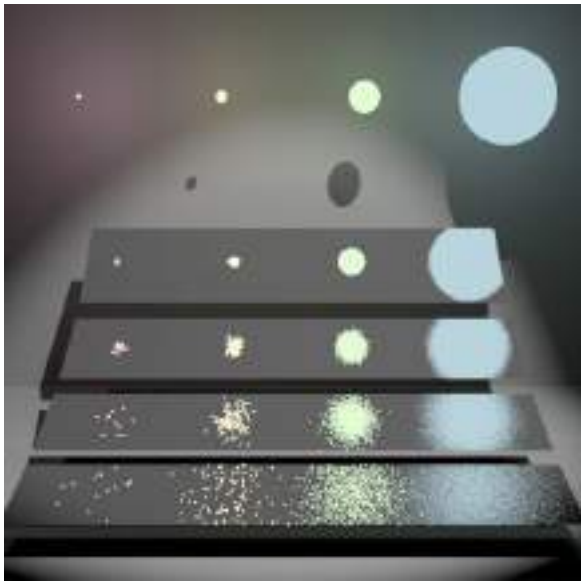
Sampling both light and BSDF

- Light sampling is not efficient for sharp material since we might miss the material lobe
- We can apply one more time multiple importance sampling to include both strategies

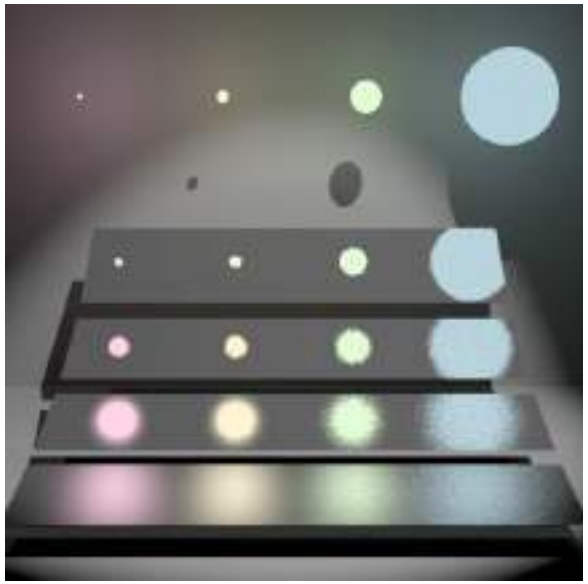
Sampling light



Sampling BSDF



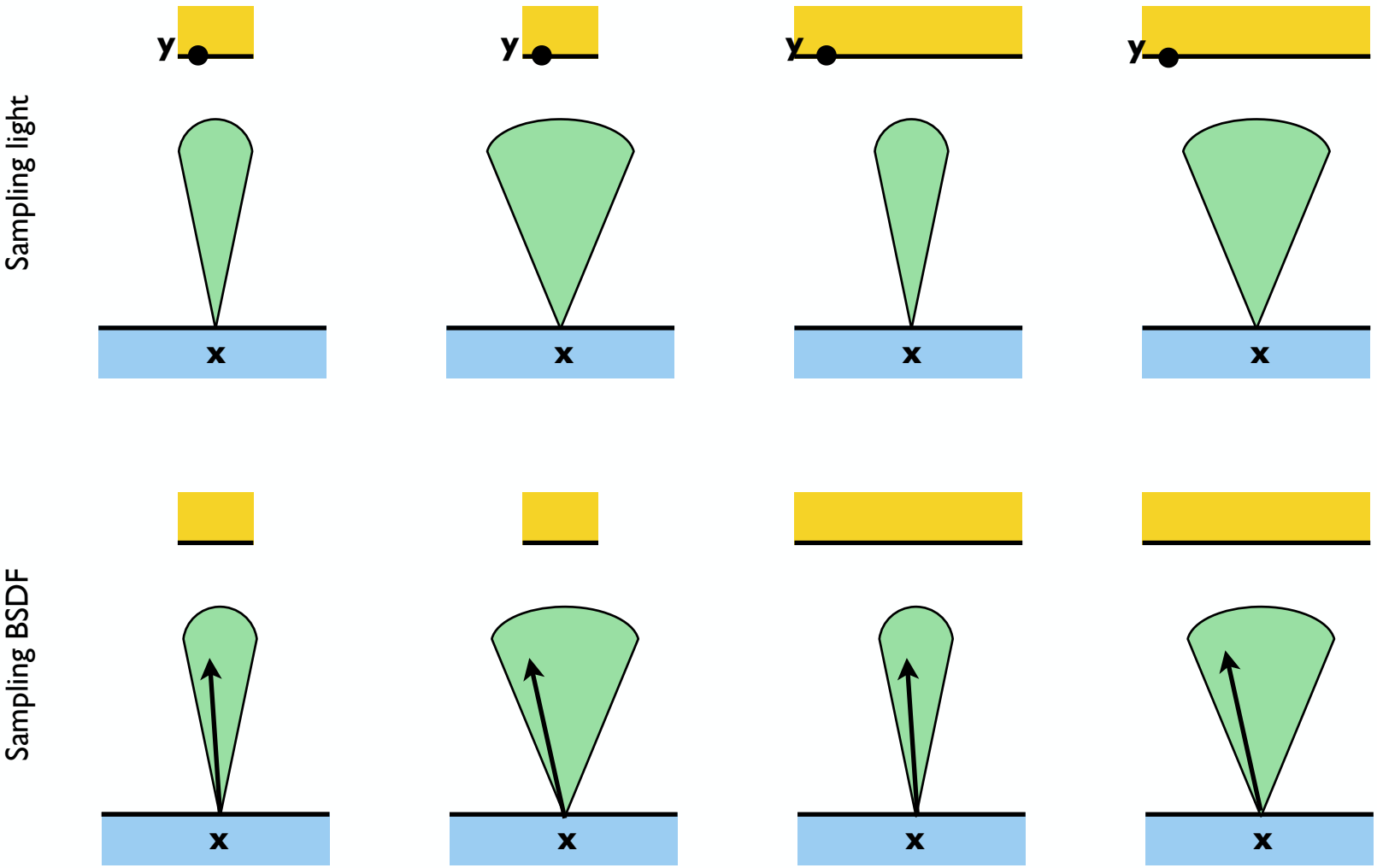
Sampling both



[Veach]

© 2020 Fabio Pellacini

Sampling both light and BSDF



© 2020 Fabio Pellacini

Sampling both light and BSDF

- We can use multiple importance sampling to sample according to both lights and BSDF, to get the benefit of both
- To do it, pick a technique at random, say with probability 0.5
- Pick a direction according to that
- Compute the final PDF as the sum of both PDFs

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \mathbf{i}_i^b \text{ or } \mathbf{i}_i^l)}$$

$$\mathbf{i}_i = \begin{cases} \mathbf{i}_i^b & r < w_b \\ \mathbf{i}_i^l & r \geq w_b = 1 - w_l \end{cases} \qquad p(\mathbf{i} | \mathbf{i}^b \text{ or } \mathbf{i}^l) = w_b p_b(\mathbf{i}) + w_l p_l(\mathbf{i})$$

© 2020 Fabio Pellacini

Sampling both light and BSDF

- The basic idea of sampling both BSDF and light can be use also for picking the direction for indirect illumination
- In fact, if we integrate BSDF and light sampling in a naive path tracer, without sampling direct explicitly, we get a simple solution that works well in a variety of cases

$$L(\mathbf{x},\mathbf{o}) \approx L_e(\mathbf{x},\mathbf{o}) + \frac{L(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i) f(\mathbf{x},\mathbf{i}_i,\mathbf{o}) |\mathbf{n}_\mathbf{x}\cdot\mathbf{i}_i|}{p(\mathbf{i}_i|\mathbf{i}_i^b \text{ or } \mathbf{i}_i^l)}$$

$$\mathbf{i}_i = \begin{cases} \mathbf{i}_i^b & r < w_b \\ \mathbf{i}_i^l & r \geq w_b = 1 - w_l \end{cases} \qquad p(\mathbf{i}|\mathbf{i}^b \text{ or } \mathbf{i}^l) = w_b p_b(\mathbf{i}) + w_l p_l(\mathbf{i})$$

© 2020 Fabio Pellacini

Path tracing – MIS

- We can implement a path tracer with MIS starting from a naive one and changing only the part when the next direction is computed
- Before rendering, we store an array of lights and their cdfs for sampling

```
void init_light(scene* scene) { // initialize lights and cdfs }
vec3f shade_mis(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    <emission>
    if(!is_delta(f))
      <handle smooth>
    else <handle delta>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


© 2020 Fabio Pellacini

```
vec3f shade_mis(scene* scene, ray3f ray) {
  auto l = vec3f{0}, w = vec3f{1};
  for(auto bounce : range(max_bounce)) {
    auto isec = intersect(scene, ray);
    if(!isec.hit) {
      l += w * eval_environment(scene, ray.d); break; }
    auto [p, n] = eval_point(isec);     // eval pos, norm
    auto [e, f] = eval_material(isec);  // eval bsdf
    auto o = -ray.d;                    // outgoing
    l += w * eval_emission(e, n, o);    // emission
    if(!is_delta(f)) { // sample smooth brdfs (fold cos into f)
      auto i = rand1f(rng) < 0.5 ?           // incoming
          sample_brdfcos(f,n,o,rand1f(rng),rand2f(rng)) :
          sample_lights(p,rand1f(rng),rand1f(rng),rand2f(rng));
      w *= eval_brdfcos(f,n,i,o) * 2 /
        (sample_brdfcos_pdf(f,n,i,o)+sample_lights_pdf(p,i));
    } else {          // sample sharp brdfs
      auto i = sample_delta(n, rand2f(rng)); // incoming
      w *= eval_delta(f,n,i,o) / sample_delta_pdf(i);
    }
    if(rand1f() >= min(1,max(w))) break;     // russian roulette
    w *= 1 / min(1,max(w));
    ray = {x, i};                           // recurse
  }
  return l;
}
```


© 2020 Fabio Pellacini

Path tracing – sampling lights

- A light is either an emissive object or an environment
- We store a separate array mostly as convenience

```
// a light is either an object or an environment and their cdf
struct light {
  object*       object = nullptr;
  environment*  environment = nullptr;
  vector<float> cdf = {};
};

// initialize an array of lights and compute cdfs
void init_light(scene* scene) {
  <handle objects>
  <handle environments>
}
```


© 2020 Fabio Pellacini

Path tracing – sampling lights

- For each object, we add a light if the object is emissive
- For sampling later, we store an sampling cdf computed as the accumulate triangle areas, which are treated as their weights

```
<handle_objects> =
  for (auto object : scene->objects) {
    if (object->material->emission == zero3f) continue;
    if (object->shape->triangles.empty()) continue;
    auto light = add_light(scene, object);
    auto shape = object->shape;
    light->cdf = vector<float>(shape->triangles.size());
    for(auto idx : range(shape->cdf.size())) {
      auto t = shape->triangles[idx];
      light->cdf[idx] = triangle_area(shape->positions[t.x],
            shape->positions[t.y], shape->positions[t.z]);
      if(idx) shape->cdf[idx] = shape->cdf[idx-1];
    }
 }
```


© 2020 Fabio Pellacini

Path tracing – sampling lights

- For each environment, we add a light if the environment is emissive
- The environment map cdf is computed as the accumulation of the pixel weights, which is the pixels' intensities scaled by their relative size

```
<handle environments> =
  for (auto environment : scene->environments) {
    if (environment->emission == zero3f) continue;
    auto light = add_light(scene, environment);
    auto texture = environment->emission_tex;
    auto size    = texture_size(texture);
    light->cdf = vector<float>(size.x * size.y);
    for (auto idx : range(light->cdf.size())) {
      auto ij = vec2i{idx % size.x, idx / size.x};
      auto theta = (ij.y + 0.5) * pi / size.y;
      auto value = lookup_texture(texture, ij);
      light->cdf[idx] = max(value) * sin(theta);
      if (idx) light->cdf[i] += light->cdf[i - 1];
    }
```


© 2020 Fabio Pellacini

Path tracing – sampling lights

- To sample a direction towards lights, we first pick a light at random

```
vec3f sample_lights(scene* scene, vec3f p,
                    float rl, float rel, vec2f ruv) {
  auto lid = sample_uniform(scene->lights.size(), rl);
  auto light = scene->lights[lid];
  if (light->object) {
    <handle object>
  } else if (light->environment) {
    <handle environment>
  }
}
```


© 2020 Fabio Pellacini

Path tracing – sampling lights

- For the object case, we then pick a random triangle and a random uv for that triangle, based on the previous sampling helpers
- From these, we compute the position on the light and set the direction accordingly

```
<handle object> =
  if (light->object) {
    auto tid   = sample_discrete_cdf(light->cdf, rel);
    auto uv    = sample_triangle(ruv);
    auto [lp, _] = eval_point(light->object, tid, uv);
    return normalize(lp - position);
  }
```


© 2020 Fabio Pellacini

Path tracing – sampling lights

- For the environment case, we then pick a texel at random, and compute the environment uv at the center of that telex, assuming that environment map pixels are small
- We then compute the direction corresponding to this uv

```
if (light->environment) {
 auto texture = light->environment->emission_tex;
 auto tid  = sample_discrete_cdf(light->cdf, rel);
 auto size = texture_size(texture);
 auto uv   = vec2f{(tid % size.x + puv.x) / size.x,
                   (tid / size.x + puv.y) / size.y};
 return transform_direction(environment->frame,
        {cos(uv.x * 2 * pi) * sin(uv.y * pi),
         cos(uv.y * pi),
         sin(uv.x * 2 * pi) * sin(uv.y * pi)});
}
```


© 2020 Fabio Pellacini

Path tracing – lights pdf

- To lights pdf for a direction is computed as the sum of each light pdf

```
float sample_lights_pdf(scene* scene, vec3f p, vec3f d) {
  auto pdf = 0.0f;
  for (auto light : scene->lights) {
    if (light->object) {
      <handle object>
    } else if(light->environment) {
      <handle object>
    }
  }
  pdf *= sample_uniform_pdf(scene->lights.size());
  return pdf;
}
```


© 2020 Fabio Pellacini

Path tracing – lights pdf

- For the object case, the pdf is computed as the pdfs of all triangles that lie along the given direction from the shaded point
- The triangle pdfs are given by the formulas before

```
<handle object> =
  // accumulate pdf over all intersections
  auto lpdf = 0.0f; auto np = p;
  for (auto bounce = 0; bounce < 100; bounce++) {
    auto isec = intersect(light->object, np, d);
    if (!isec) break;
    // compute pdf of this triangle
    auto [lp, ln] = eval_point(isec);
    // prob triangle * area triangle = area triangle mesh
    auto area = light->cdf.back();
    lpdf += distance_squared(lp, p) / (abs(dot(ln, d)) * area);
    np = lp + direction * 1e-3f; // continue ray
  }
  pdf += lpdf;
```


© 2020 Fabio Pellacini

Path tracing – lights pdf

- For the environment case, the pdf is computed as the pdf of choosing that pixel over the solid angle of the selected pixel

```
<handle environment> =
  auto texture = light->environment->emission_tex;
  auto size = texture_size(texture);
  auto wl = transform_direction(
            inverse(light->environment->frame), direction);
  auto texcoord = vec2f{atan2(wl.z, wl.x) / (2 * pif),
                  acos(clamp(wl.y, -1.0f, 1.0f)) / pif};
  if (texcoord.x < 0) texcoord.x += 1;
  auto i    = clamp((int)(texcoord.x * size.x), 0, size.x - 1);
  auto j    = clamp((int)(texcoord.y * size.y), 0, size.y - 1);
  auto prob = sample_discrete_cdf_pdf(light->cdf, j*size.x+i) /
                    light->cdf.back();
  auto angle = (2 * pif / size.x) * (pif / size.y) *
                     sin(pif * (j + 0.5f) / size.y);
  pdf += prob / angle;
```


© 2020 Fabio Pellacini

Path tracing – lights pdf

- Overall we made use a few simple support functions

```
// sample an index with uniform probability
int sample_uniform(int size, float r) {
  return clamp((int)(r * size), 0, size - 1); }
float sample_uniform_pdf(int size) {
  return 1.0 / (float)size; }
// sample an index with the given cdf
int sample_discrete_cdf(vector<float> cdf, float r) {
  r = clamp(r * cdf.back(), 0, cdf.back() - 0.00001);
  auto idx = (int)(std::upper_bound(cdf.begin(),
                  cdf.end(), r) - cdf.begin());
  return clamp(idx, 0, cdf.size() - 1); }
float sample_discrete_cdf_pdf(vector<float> cdf, int idx) {
  if (idx == 0) return cdf.at(0);
  else return cdf.at(idx) - cdf.at(idx - 1); }
// sample a triangle uv
vec2f sample_triangle(vec2f ruv) {
  return {1 - sqrt(ruv.x), ruv.y * sqrt(ruv.x)}; }
```


© 2020 Fabio Pellacini

Materials

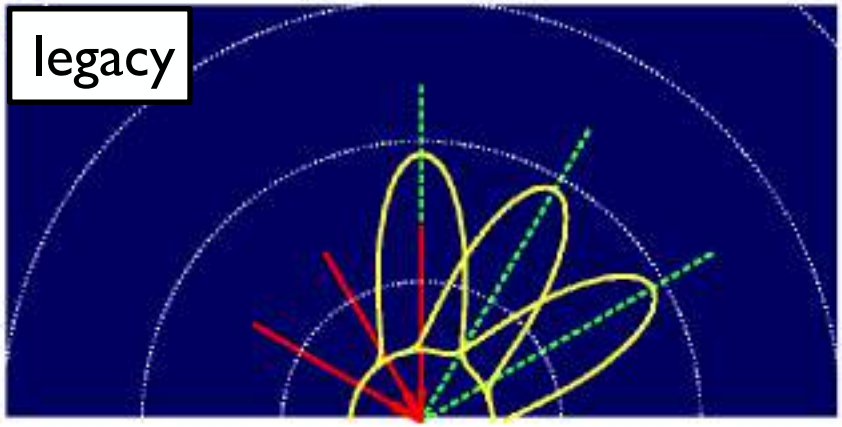
© 2020 Fabio Pellacini

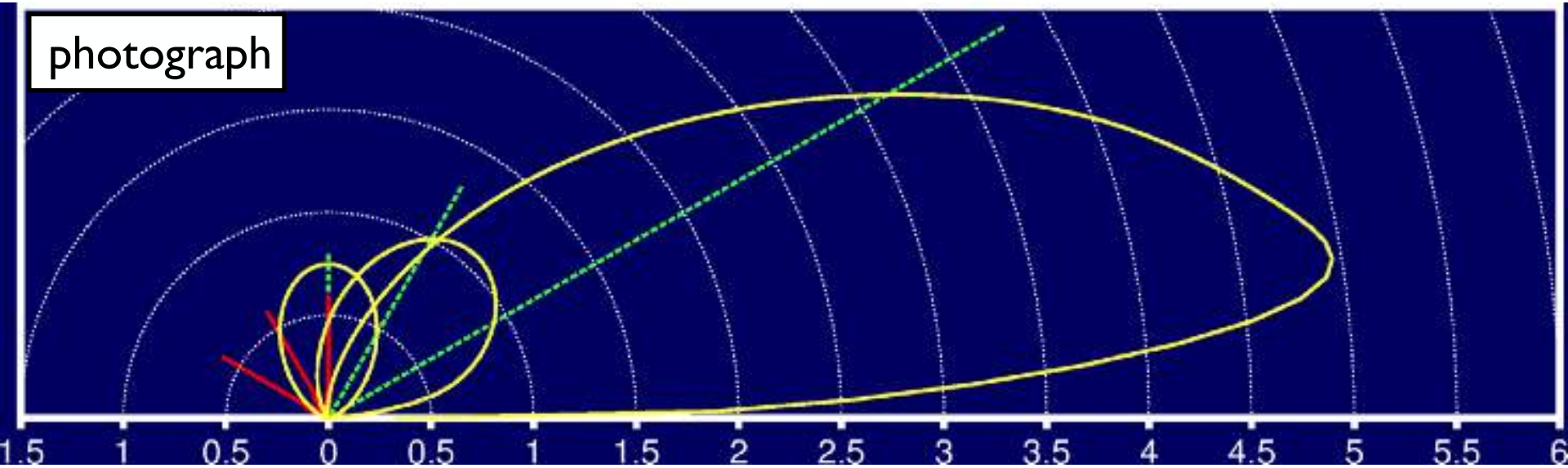
Legacy vs physically-based

[Lafortune et al]

© 2020 Fabio Pellacini

Legacy vs physically-based





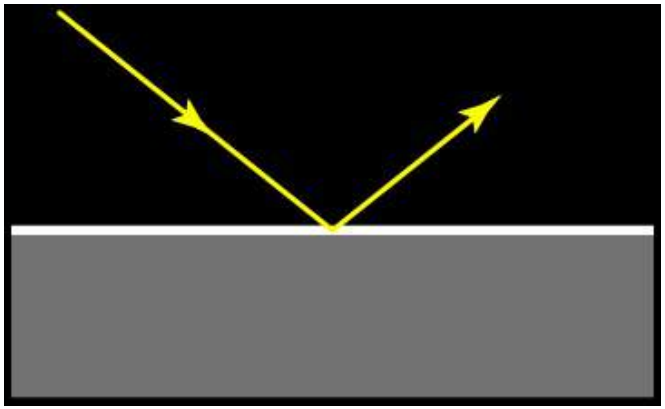
© 2020 Fabio Pellacini

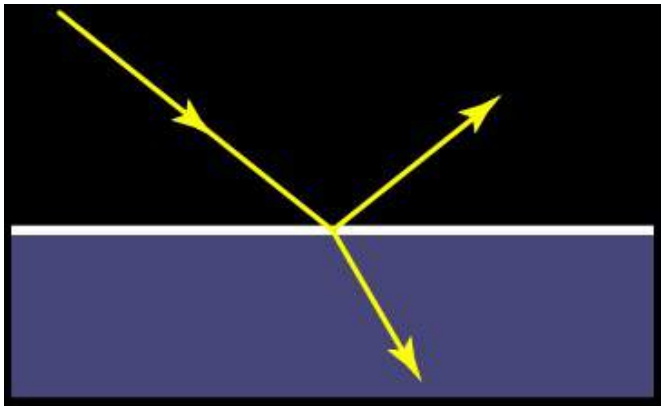
Materials

- We'll focus on BSDF and in particular in covering opaque materials
- Different BSDFs comes from different optical behavior and "structure" of the surface
- In theory, most materials would need an entirely different BSDF to account for their peculiarities
  - not practical since editing and exchanging them becomes hard
- In practice, use one "good-enough" physical model for most cases and custom materials only when absolutely necessary

© 2020 Fabio Pellacini

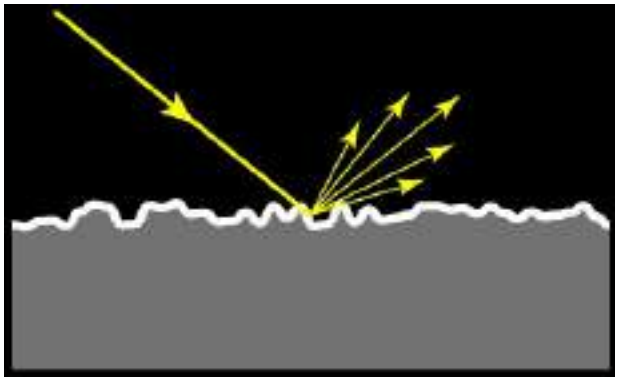
Smooth surfaces

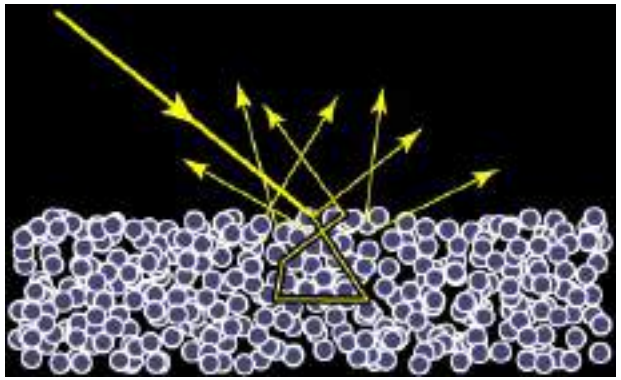




© 2020 Fabio Pellacini

Rough surfaces



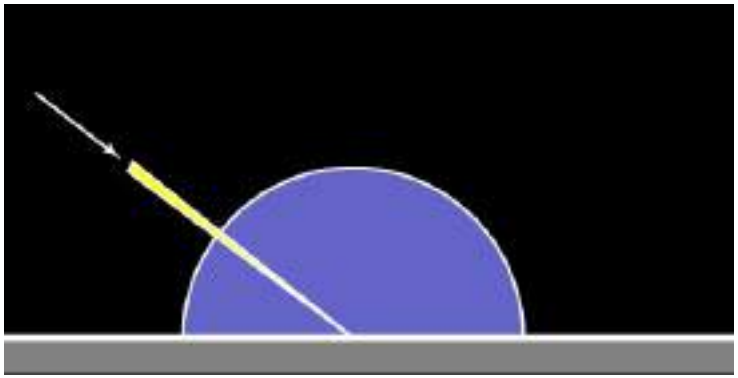


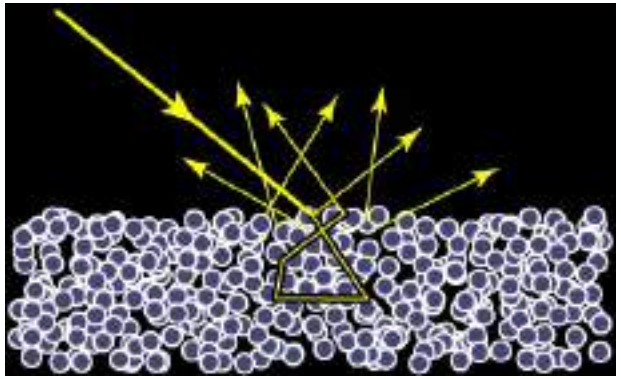
© 2020 Fabio Pellacini

Diffuse reflection

- Diffuse reflection from a completely matte surface
- Light is scattered in all direction uniformly
- BRDF is a constant
- For energy conservation, k_d is in [0,1] and normalized by *pi*

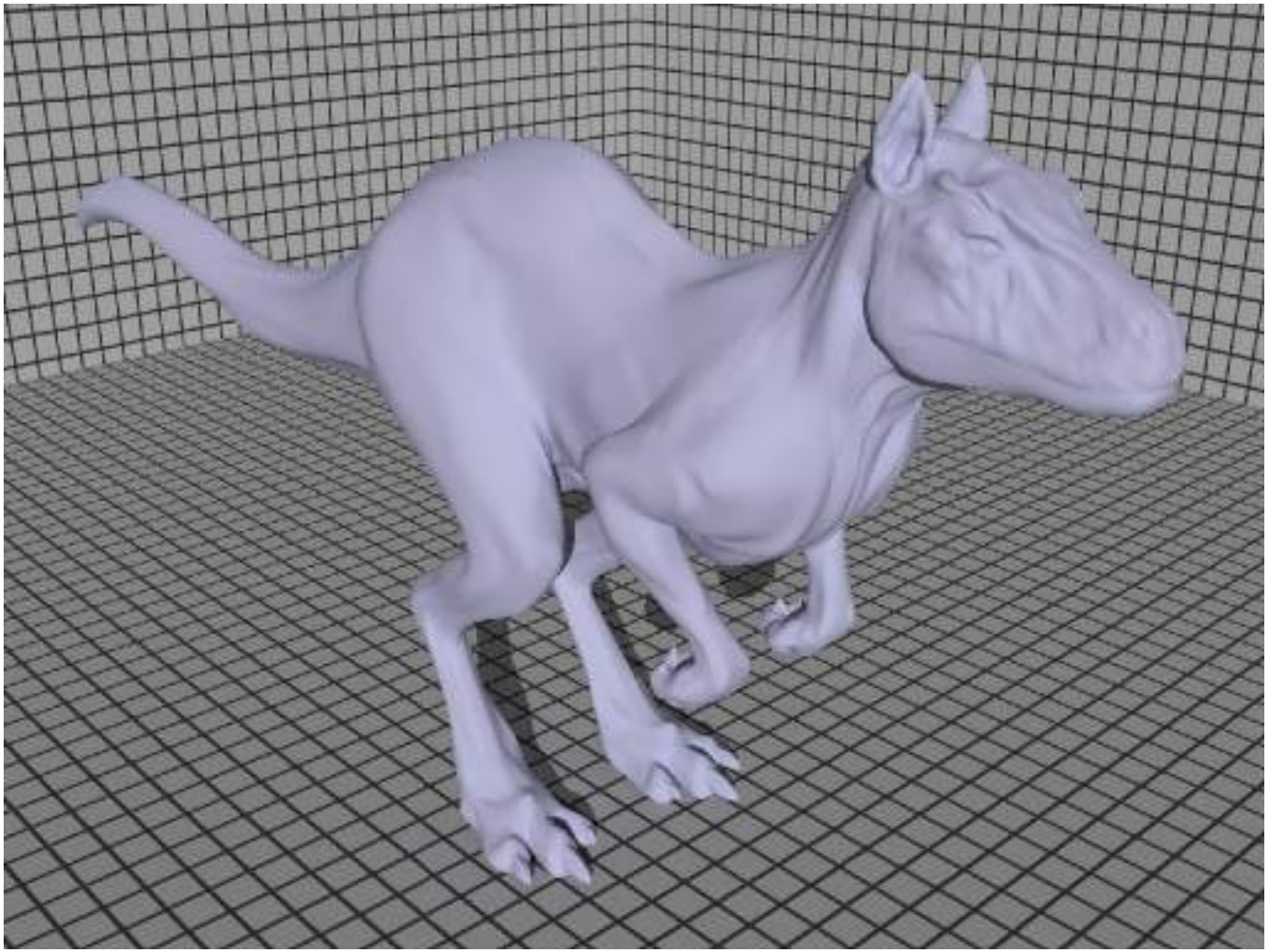
$$f_d(\mathbf{i}, \mathbf{o}) = \frac{k_d}{\pi}$$





© 2020 Fabio Pellacini

Diffuse reflection



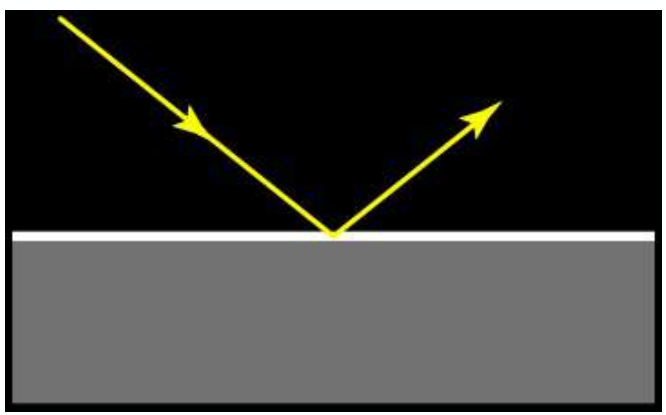
[Pharr et al./pbrt]

© 2020 Fabio Pellacini

Ideal reflection

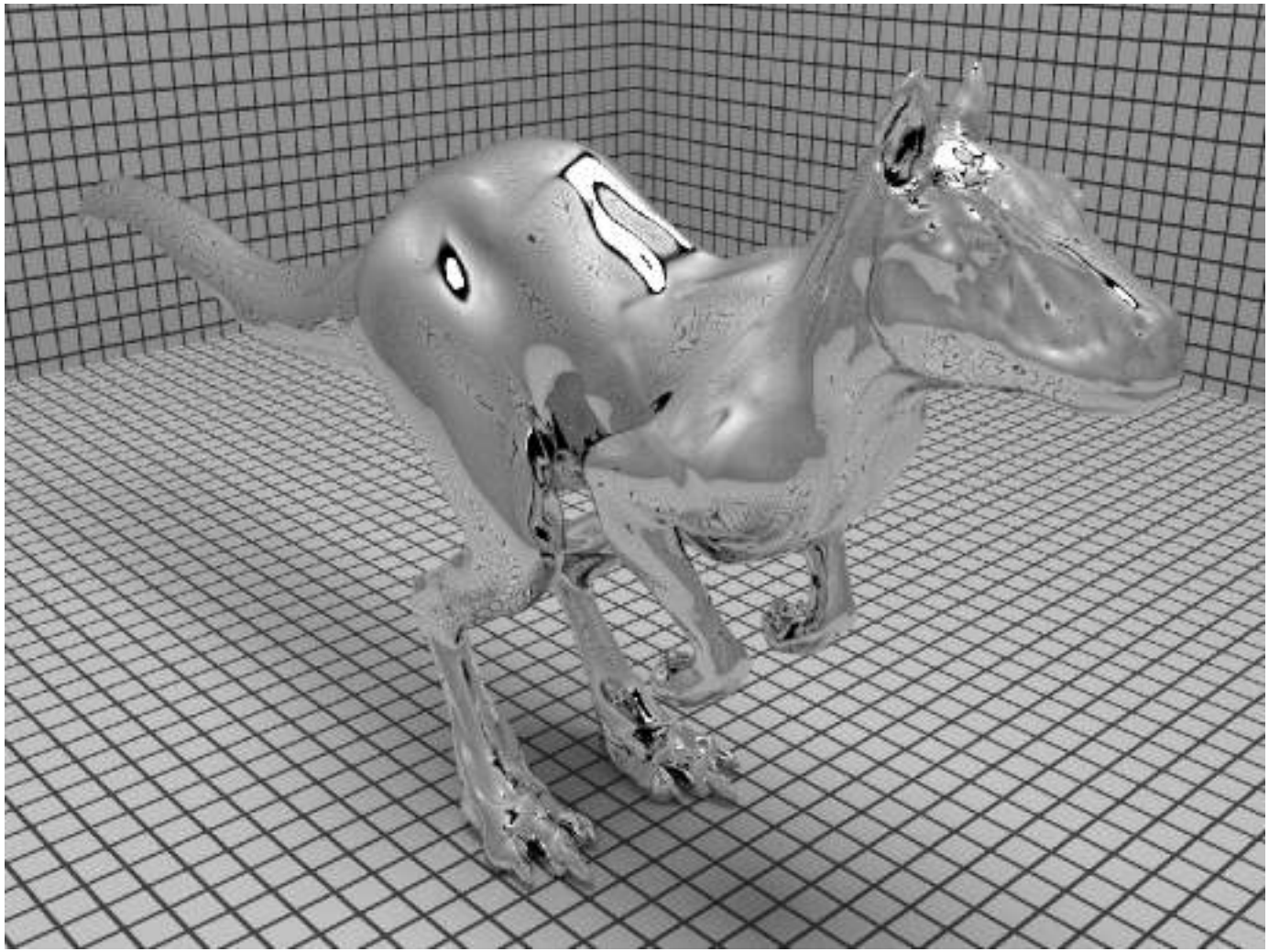
- Ideal surface with no imperfection made of an ideal conductor
- Light is scattered entirely along the mirror direction
- BRDF is a "delta function" — informally, zero everywhere except at one point, and with finite integral

$$f_r(\mathbf{i}, \mathbf{o}, \mathbf{n}) = F(\mathbf{o}, \mathbf{n}) \frac{\delta(\mathbf{i}, \mathbf{r})}{|\mathbf{n} \cdot \mathbf{i}|} \quad \mathbf{r} = -\mathbf{o} + 2(\mathbf{n} \cdot \mathbf{o})\mathbf{n}$$



© 2020 Fabio Pellacini

Ideal reflection



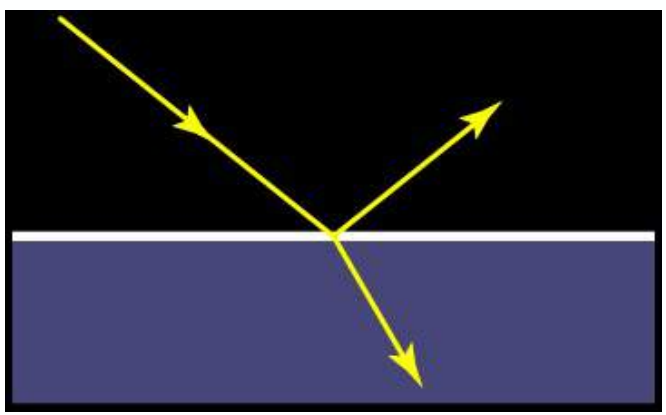
[Pharr et al./pbrt]

© 2020 Fabio Pellacini

Ideal refraction

- Ideal surface made of a dialectic has both reflection and refraction
- Light is scattered in two directions, above and below the surface
- BRDF refraction is a "delta function" around the refracted direction

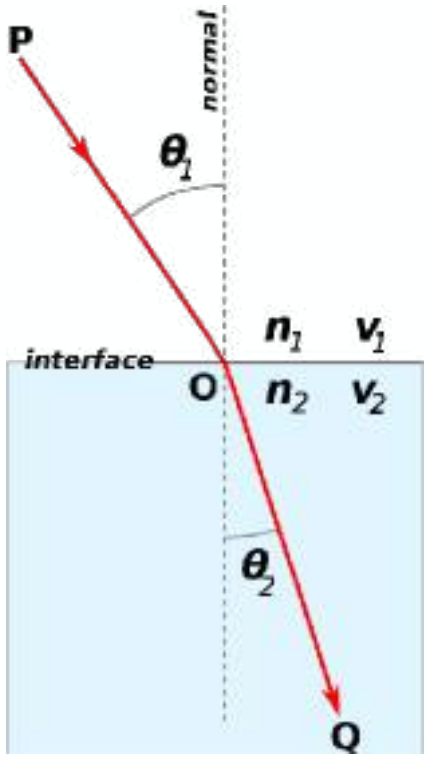
$$f_t(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{1}{\eta^2}(1 - F(\mathbf{o},\mathbf{n}))\frac{\delta(\mathbf{i},\mathbf{t})}{|\mathbf{n}\cdot\mathbf{i}|} \quad \mathbf{t} = -\frac{1}{\eta}\mathbf{o} + \left[\frac{1}{\eta}(\mathbf{n}\cdot\mathbf{o}) - \sqrt{1 + \frac{1}{\eta^2}(|\mathbf{n}\cdot\mathbf{o}|^2 - 1)}\right]\mathbf{n}$$



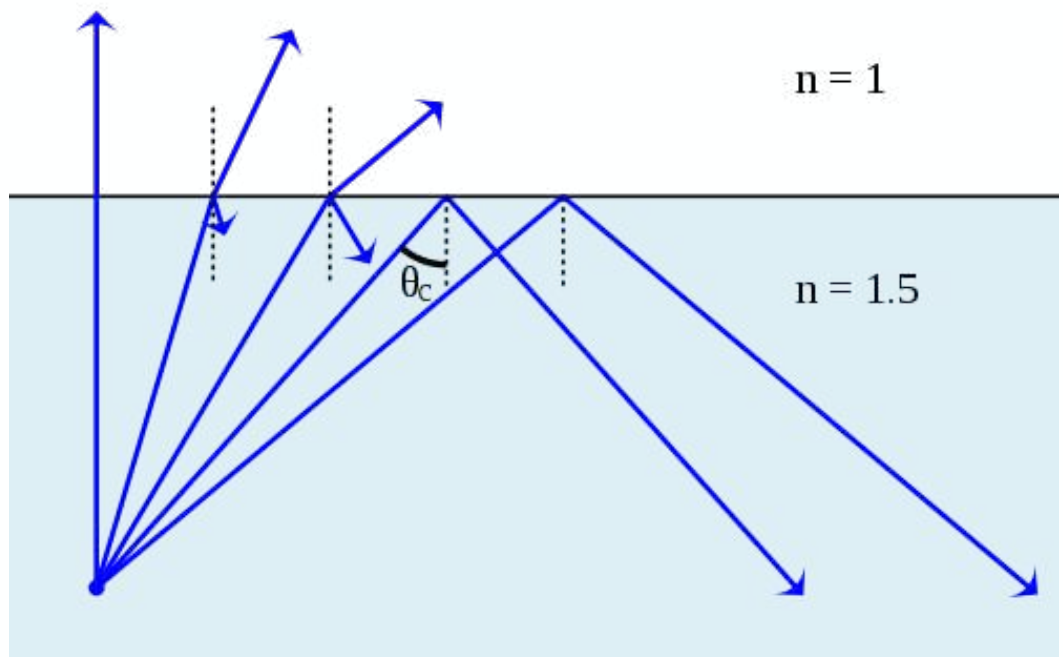
$$\eta = \frac{\eta_i}{\eta_o}$$

© 2020 Fabio Pellacini

Ideal refraction



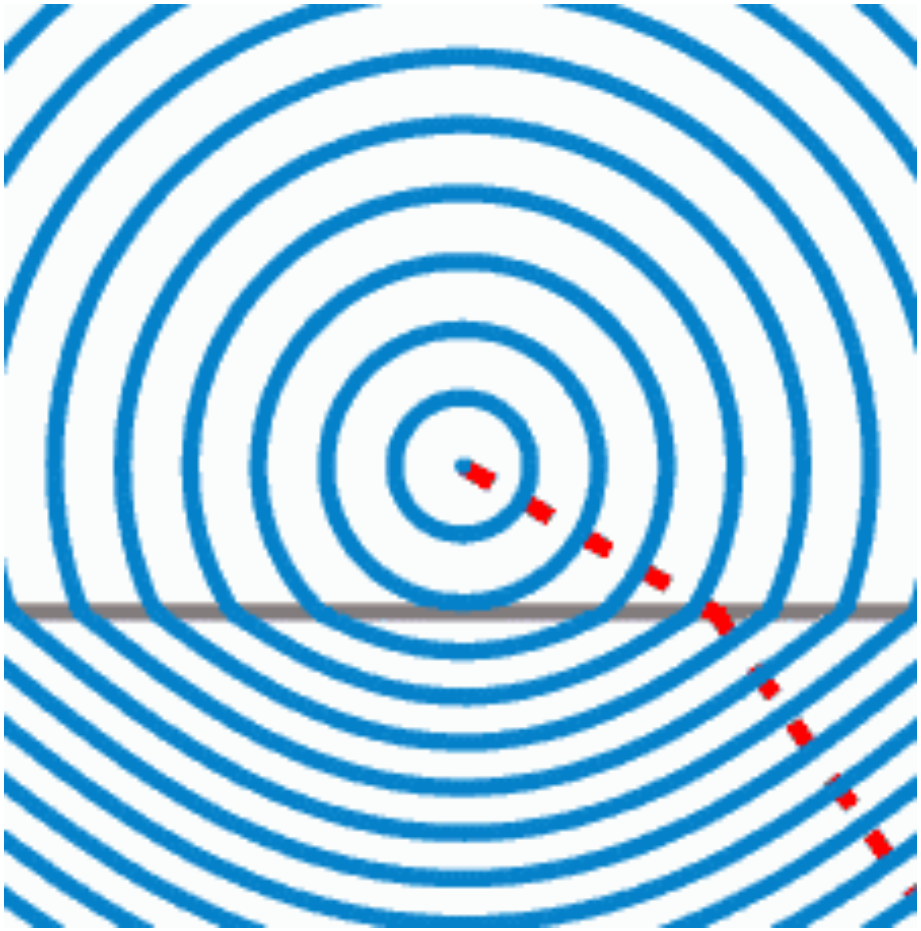
$$\mathbf{t} = -\frac{1}{\eta}\mathbf{o} + \left[\frac{1}{\eta}(\mathbf{n}\cdot\mathbf{o}) - \sqrt{1 + \frac{1}{\eta^2}(|\mathbf{n}\cdot\mathbf{o}|^2 - 1)}\right]\mathbf{n}$$



[Wikipedia]

© 2020 Fabio Pellacini

Ideal refraction

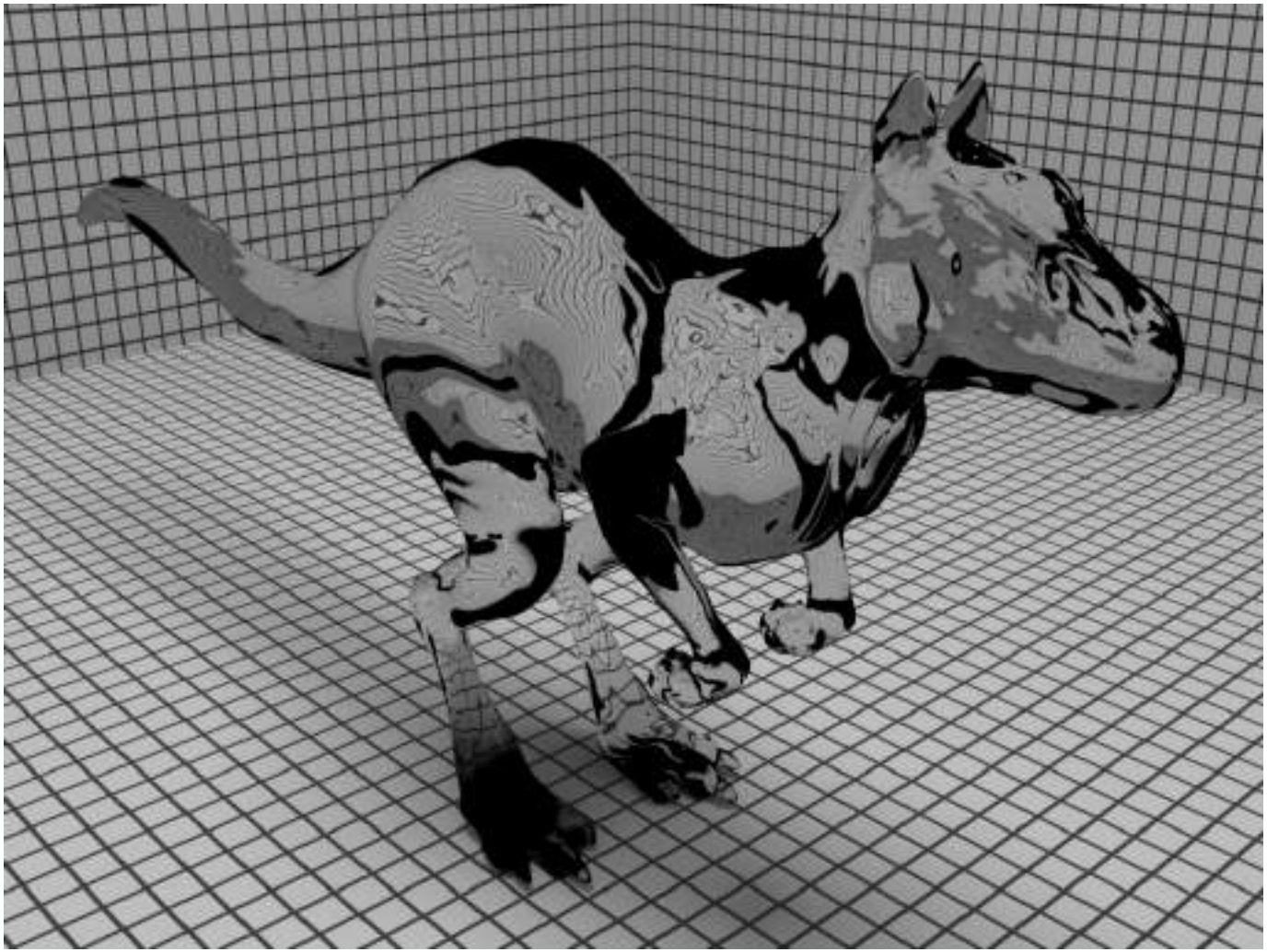


$$f_t(\mathbf{i}, \mathbf{o}, \mathbf{n}) = \frac{1}{\eta^2}(1 - F(\mathbf{o}, \mathbf{n}))\frac{\delta(\mathbf{i}, \mathbf{t})}{|\mathbf{n} \cdot \mathbf{i}|}$$

[Wikipedia]

© 2020 Fabio Pellacini

Ideal refraction

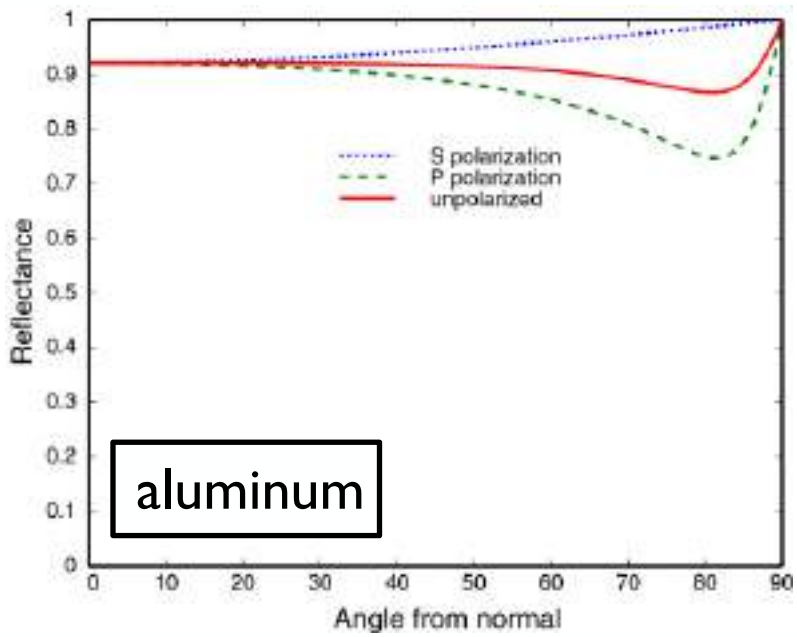


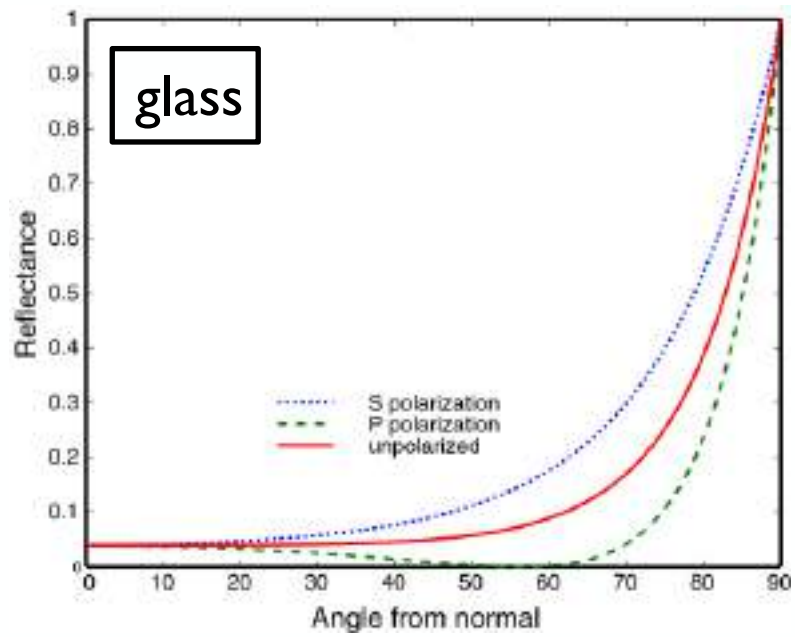
[Pharr et al./pbrt]

© 2020 Fabio Pellacini

Fresnel reflection and transmission

- Fresnel term: determines k_r and k_t from intrinsic material properties
- Conductors: small variation from surface color to 1
- Dielectrics: drastic change from small values (~0.04) to 1
- Depends on wavelength





[Westin]

© 2020 Fabio Pellacini

Schlick's Fresnel approximation

- Real-time graphics uses an approximation for Fresnel due to Schlick
- Mostly valid for dialectics similar to glass
- Less valid for metals, but hard to see the difference
- In general, better to use more appropriate approximations for dielectrics to capture TIR

$$F_s(\mathbf{o}, \mathbf{n}) = F_0 + (1 - F_0)(1 - \mathbf{o} \cdot \mathbf{n})^5 \quad F_0 = \frac{(\eta - 1)^2}{(\eta + 1)^2}$$

© 2020 Fabio Pellacini

Fresnel formulas

- Dielectric Fresnel

$$F_d(\mathbf{o}, \mathbf{n}) = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right)$$

$$g = \sqrt{\eta^2 - 1 + c^2} \quad \text{and} \quad c = |\mathbf{o} \cdot \mathbf{n}|$$

- Conductor Fresnel: too complex to write here
  - https://seblagarde.wordpress.com/2013/04/29/memo-on-fresnel-equations/
  - http://www.pbr-book.org/3ed-2018/Reflection_Models/Specular_Reflection_and_Transmission.html

© 2020 Fabio Pellacini

Fresnel term

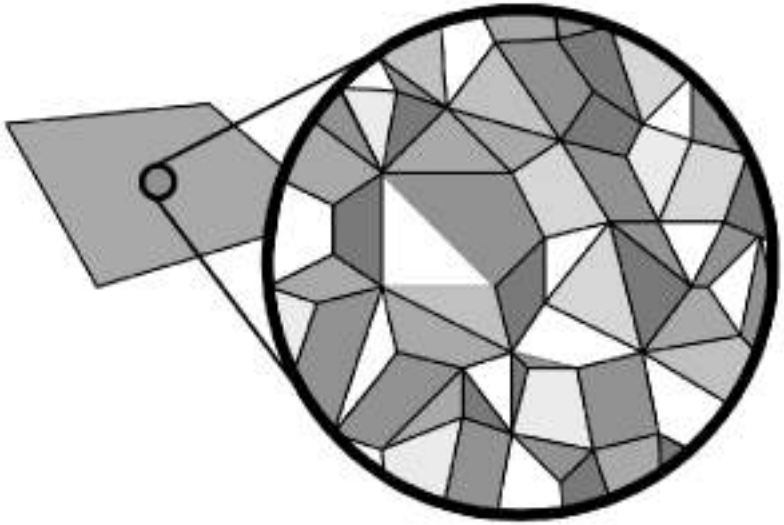
[Wikipedia]

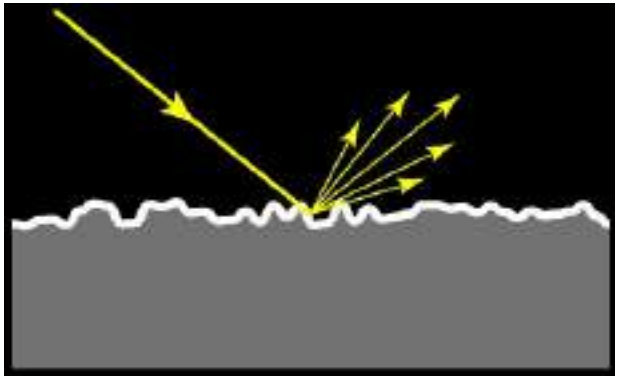
© 2020 Fabio Pellacini

Microfacet reflection

- Surface modeled as a collection of randomly oriented microfacet
- When light hits a surface, it hits one microfacet
- BRDF is the "average" value of all these reflection events
  - reflection centered around the bisector h
- See "Microfacet Models for Refraction through Rough Surfaces"

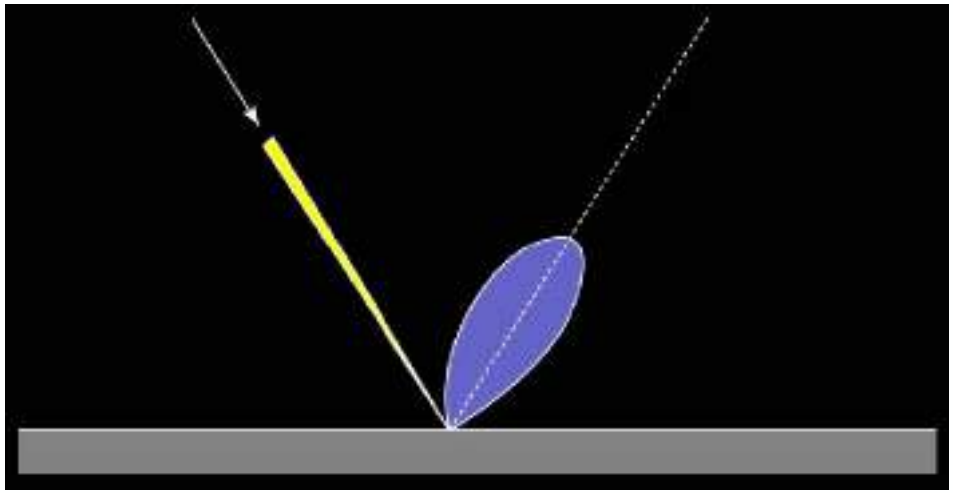
[Westin]

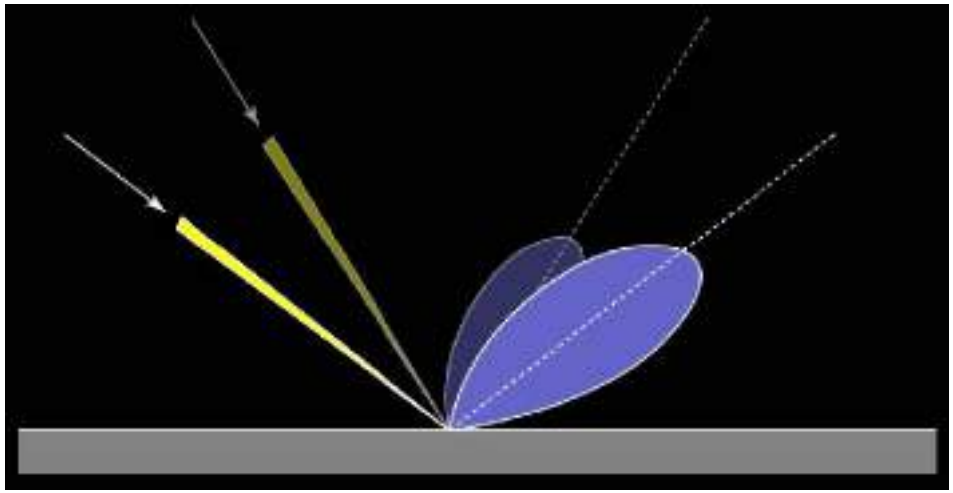


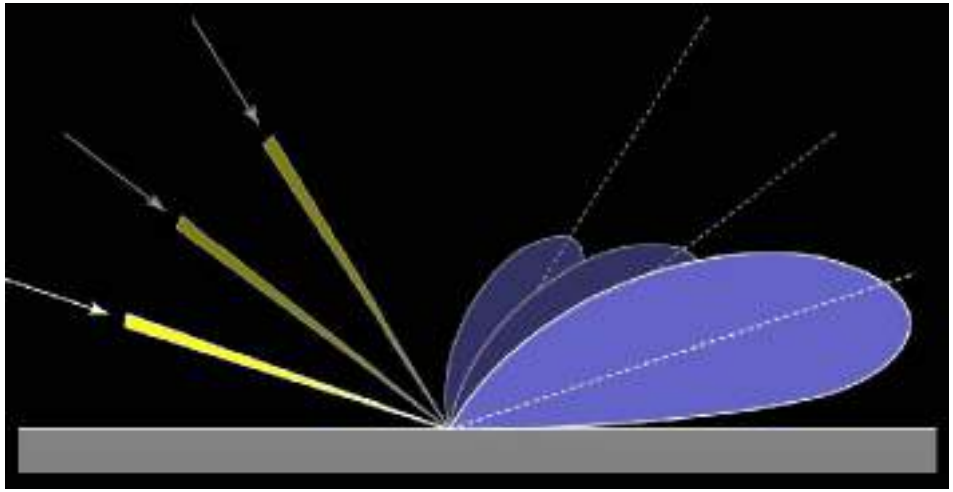


© 2020 Fabio Pellacini

Microfacet reflection





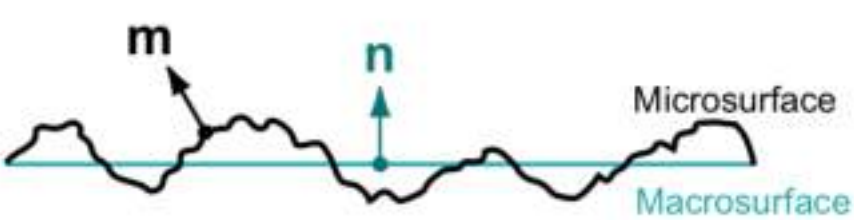


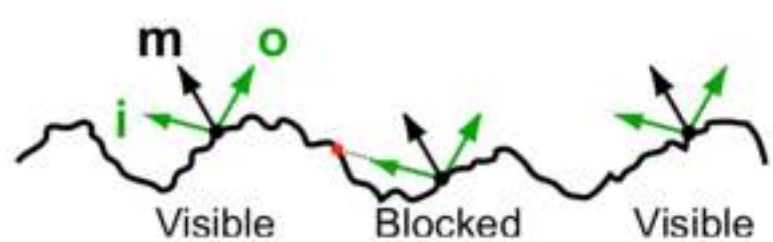
© 2020 Fabio Pellacini

Microfacet reflection

- Normalized product of three terms: fresnel *F*, microfacet distribution *D* and shadow-masking term *G*
- Fresnel *F* accounts for difference at grazing angle
- Distribution *D* describes the statistical distribution of microfacets
- Shadow-making *G* corrects for facets not visible from light or view

$$f_r(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{F(\mathbf{o},\mathbf{n})D(\mathbf{h},\mathbf{n})G(\mathbf{i},\mathbf{o},\mathbf{n})}{4|\mathbf{n}\cdot\mathbf{i}||\mathbf{n}\cdot\mathbf{o}|} \qquad \mathbf{h} = \frac{\mathbf{i}+\mathbf{o}}{|\mathbf{i}+\mathbf{o}|}$$





[Walter et al.]

© 2020 Fabio Pellacini

Microfacet reflection

- Different choices for *D*, mostly quite similar
- Blinn-Phong: simple, but lacks physical basis

$$D_p(\mathbf{h}, n) = \frac{n+2}{2\pi}(\mathbf{n}\cdot\mathbf{h})^n \qquad D_p(\mathbf{h}, \alpha) = \frac{1}{\pi\alpha^2}(\mathbf{n}\cdot\mathbf{h})^{\frac{2}{\alpha^2}-2}$$

- Beckman: first distribution used in graphics

$$D_b(\mathbf{h}, \alpha) = \frac{1}{\pi\alpha^2(\mathbf{n}\cdot\mathbf{h})^4}\exp\left(\frac{(\mathbf{n}\cdot\mathbf{h})^2 - 1}{\alpha^2(\mathbf{n}\cdot\mathbf{h})^2}\right)$$

- GGX: currently the most used one – slightly better fit to real data

$$D_g(\mathbf{h}, \alpha) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2-1)+1)^2}$$

© 2020 Fabio Pellacini

Microfacet reflection

- Different choices for *G*, quite a bit different – use Smith formulation

$$G(\mathbf{i}, \mathbf{o}, \mathbf{n}) = G_1(\mathbf{i}, \mathbf{n}) G_1(\mathbf{o}, \mathbf{n})$$

- Blinn-Phong: approximate as Beckmann
- Beckmann: slow analytic solution, better to use approximation

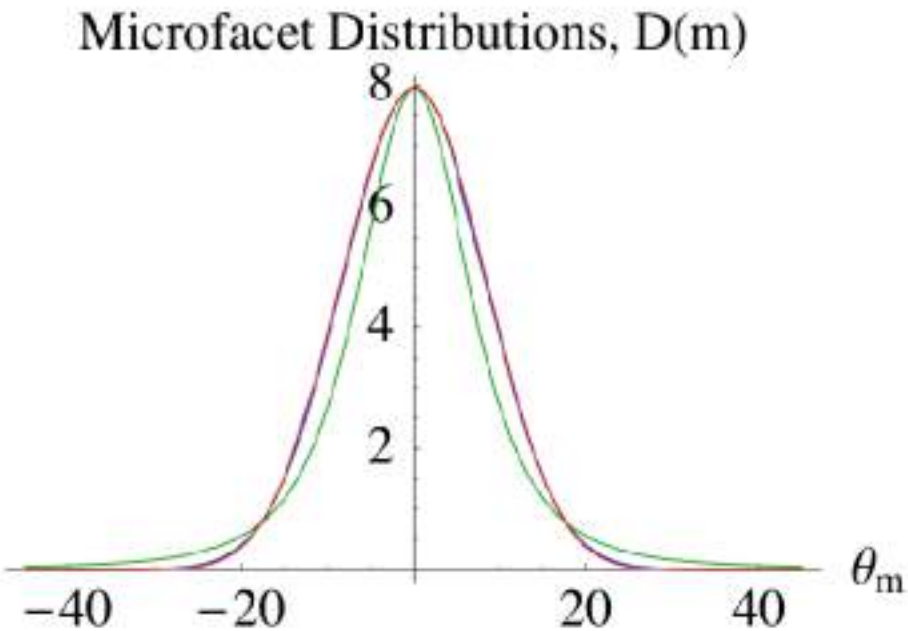
$$G_{1,b}(\mathbf{v}, \mathbf{n}) \approx \begin{cases} \frac{3.535a + 2.181a^2}{1 + 2.276a + 2.577a^2} & a < 1.6 \\ 1 & \text{otherwise} \end{cases} \qquad a = \frac{|\mathbf{n} \cdot \mathbf{v}|}{\alpha_b \sqrt{1 - |\mathbf{n} \cdot \mathbf{v}|^2}}$$

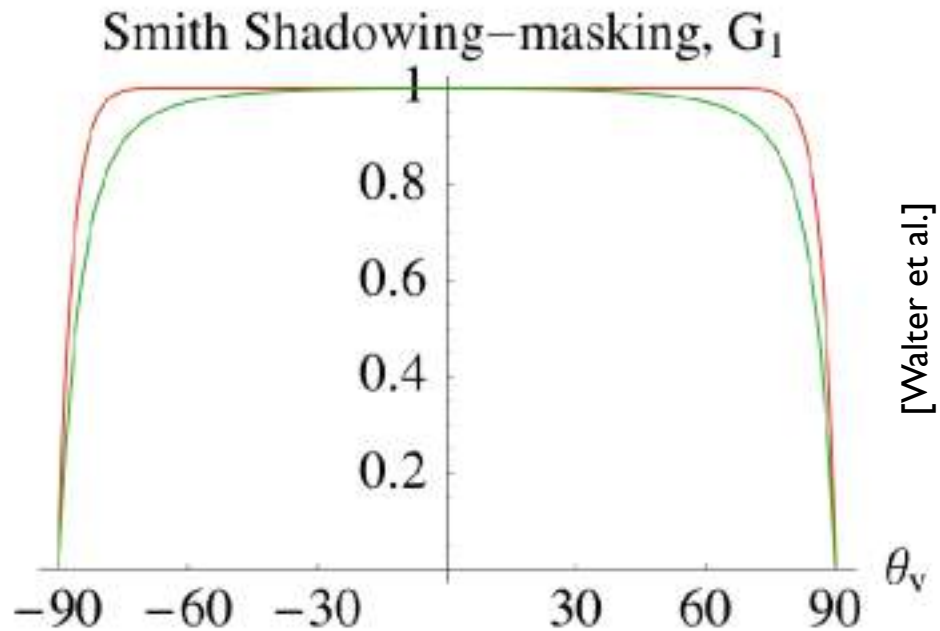
- GGX: analytic solution

$$G_{1,g}(\mathbf{v}, \mathbf{n}) = \frac{2|\mathbf{n} \cdot \mathbf{v}|}{|\mathbf{n} \cdot \mathbf{v}| + \sqrt{\alpha_g^2 + (1 - \alpha_g^2)|\mathbf{n} \cdot \mathbf{v}|^2}}$$

© 2020 Fabio Pellacini

Microfacet reflection

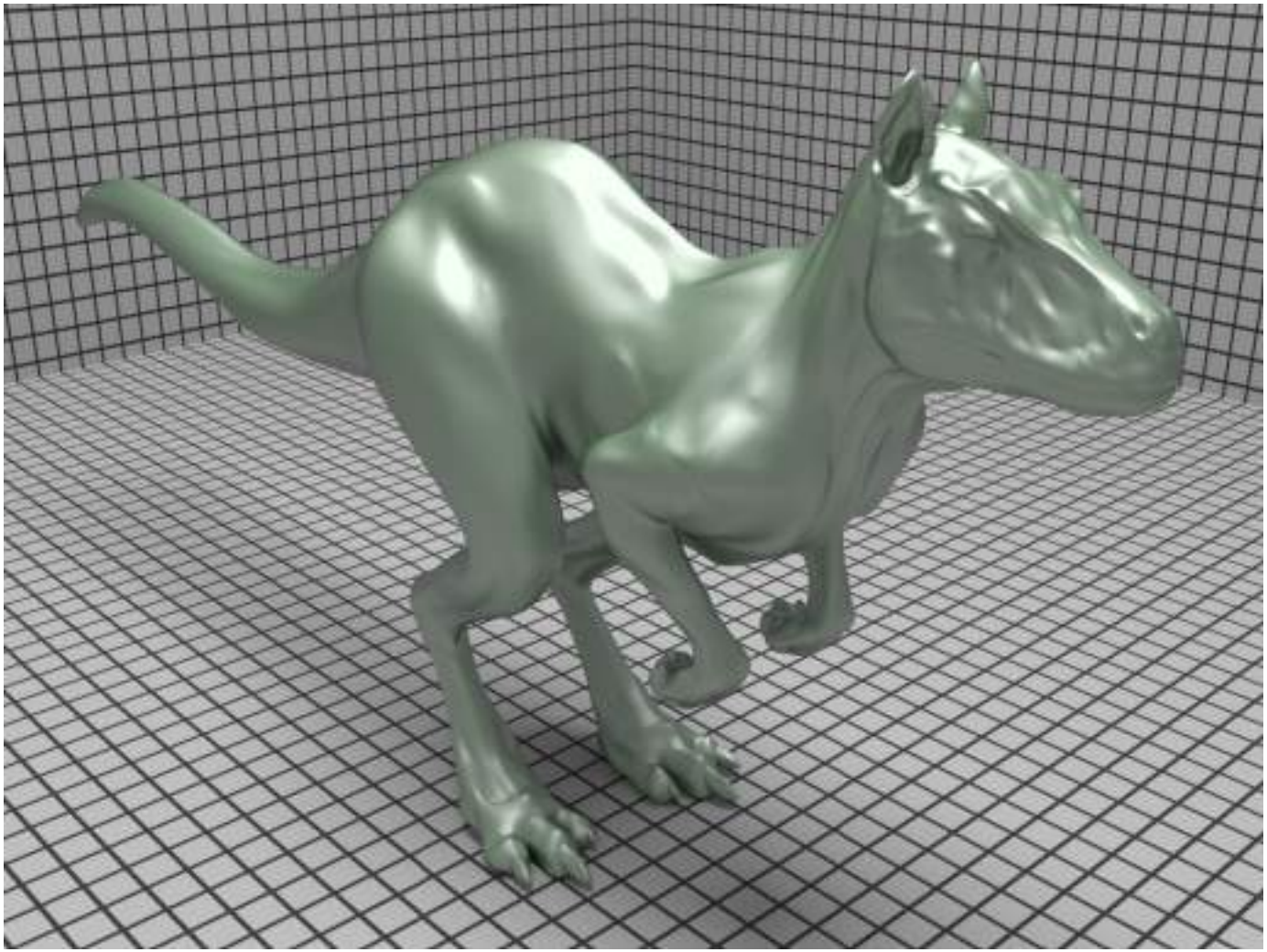




[Walter et al.]

© 2020 Fabio Pellacini

Microfacet reflection



[Pharr et al./pbrt]

© 2020 Fabio Pellacini

Microfacet transmission

- Derived from previous ideas while correctly accounting for change of variable when moving below the surface
  - Here we write equations using the relative index of refraction which is eta on entering and 1/eta on exiting

$$f_t(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{|\mathbf{h}\cdot\mathbf{i}|\ |\mathbf{h}\cdot\mathbf{o}|}{|\mathbf{n}\cdot\mathbf{i}|\ |\mathbf{n}\cdot\mathbf{o}|}\frac{(1-F(\mathbf{o},\mathbf{h}))D(\mathbf{h},\mathbf{n})G(\mathbf{i},\mathbf{o},\mathbf{n})}{(\eta(\mathbf{h}\cdot\mathbf{i})+(\mathbf{h}\cdot\mathbf{o}))^2}$$

$$\mathbf{h} = \frac{-\eta\mathbf{i}+\mathbf{o}}{|-\eta\mathbf{i}+\mathbf{o}|}\mathrm{sign}(\mathbf{n}\cdot\mathbf{o})$$

© 2020 Fabio Pellacini

Generic materials

- These scattering behaviors can be combined together to *fit* data from real-world surfaces by using weighted sums of lobes

$$f = \sum_i k_i f_i$$

- Polished/rough glass: combine reflection and transmission terms
- Plastics: weighted sum of diffuse and specular reflections
  - Accounting for Fresnel still an open problem here, but we can use a simple solution that conserve energy

$$f_{plastic} = k_d(1 - F(\mathbf{n} \cdot \mathbf{o}))f_d + k_s f_s$$

© 2020 Fabio Pellacini

Sampling BSDF-cosine

- When picking directions for hemispherical sampling, we want to sample a function that is close to the integrand
- Often, best we can do is to sample the BSDF scaled by the cosine
- For mirror reflection and transmission, we only have one direction
- For diffuse reflection, we use the cosine sampling presented previously
- For convenience, we can rewrite that sampling in spherical coordinates

$$\mathbf{i}^l = (\phi, \theta) = (2\pi r_1, \mathrm{acos}(r_2))$$

$$p(\mathbf{i}^l) = \frac{\mathbf{i}^l_z}{\pi}$$

© 2020 Fabio Pellacini

Sampling BSDF-cosine

- For microfacet BRDFs, we sample the *D* term since this is dominant
- We proceed by picking a half-vector **h** according to *D* as

$$\mathbf{h}_p^l = \left(2\pi r_1, \operatorname{acos}\left(r_2^{\alpha/2}\right)\right)$$

$$\mathbf{h}_b^l = \left(2\pi r_1, \operatorname{atan}\left(\sqrt{-\alpha^2 \log(1 - r_2)}\right)\right)$$

$$\mathbf{h}_g^l = \left(2\pi r_1, \operatorname{atan}\left(\frac{\alpha\sqrt{r_2}}{\sqrt{1 - r_2}}\right)\right)$$

$$p(\mathbf{h}) = D(\mathbf{h})|\mathbf{n} \cdot \mathbf{h}|$$

© 2020 Fabio Pellacini

Sampling BSDF-cosine

- For reflection, we derive the incoming angle from the half-vector and correct the pdf for the change of variable as

$$\mathbf{i} = -\mathbf{o} + (\mathbf{h} \cdot \mathbf{o})\mathbf{h} \qquad p(\mathbf{i}) = \frac{D(\mathbf{h}, \mathbf{n})|\mathbf{n} \cdot \mathbf{h}|}{4|\mathbf{h} \cdot \mathbf{i}|}$$

- For refraction, the incoming angle and pdf are written as

$$\mathbf{i} = -\frac{1}{\eta}\mathbf{o} + \left[\frac{1}{\eta}(\mathbf{h} \cdot \mathbf{o}) - \sqrt{1 + \frac{1}{\eta^2}(|\mathbf{h} \cdot \mathbf{o}|^2 - 1)}\right]\mathbf{h}$$

$$p(\mathbf{i}) = \frac{D(\mathbf{h}, \mathbf{n})|\mathbf{n} \cdot \mathbf{h}|}{4|\mathbf{h} \cdot \mathbf{i}|} \frac{|\mathbf{h} \cdot \mathbf{o}|}{(\eta(\mathbf{h} \cdot \mathbf{i}) + (\mathbf{h} \cdot \mathbf{o}))^2}$$

© 2020 Fabio Pellacini

Sampling BSDF-cosine

- For BSDF that are sums of lobes, we use multiple importance sampling
- We pick the lobe at random where lobes treated as a discreet distribution, with weights set as the maximum of the lobe weight
- We then sample that lobe
- The final PDF is the sum of PDFs

$$f = \sum_i k_i f_i \qquad w_i = \frac{\max(k_i)}{\sum_i \max(k_i)}$$

$$\mathbf{i} = \mathbf{i}_i \text{ if } \sum_j^i w_j > r \qquad p(\mathbf{i}) = \sum_i p_i(\mathbf{i})$$

© 2020 Fabio Pellacini

Diffuse reflection

- For diffuse reflection, the implementation is straightforward
- We always need to check in which hemisphere we are

```
vec3f eval_diffuse(vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return vec3f{0};
  return vec3f{1} / pi * dot(n, i);
}

vec3f sample_diffuse(vec3f n, vec3f o, vec2f rn) {
  if (dot(n, o) <= 0) return vec3f{0};
  return sample_hemisphere_cos(n, rn);
}

float sample_diffuse_pdf(vec3f n, vec3f o, vec3 i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return 0;
  return sample_hemisphere_cos_pdf(n, i);
}
```


© 2020 Fabio Pellacini

Diffuse reflection

- We base the implementation on hemispherical sampling functions

```
vec3f sample_hemisphere_cos(vec3f n, vec2f ruv) {
  auto il = vec3f{sqrt(1 - ruv.y*ruv.y) * cos(2*pi*ruv.x),
                  sqrt(1 - ruv.y*ruv.y) * sin(2*pi*ruv.x),
                  sqrt(ruv.y)};
  return basis_framez(n) * il;
}

float sample_hemisphere_cos_pdf(vec3f n, vec3f i) {
  auto cosw = dot(n, d);
  return (cosw <= 0) ? 0 : cosw / pi;
}
```


© 2020 Fabio Pellacini

Microfacet reflection

- For micro facet reflections, our implementation follows the math formulation precisely

```
vec3f eval_specular(float ior, float a,
    vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return vec3f{0};
  auto h = normalize(incoming + outgoing);
  auto F = fresnel_dielectric(ior, h, i);
  auto D = microfacet_distribution(a, n, h);
  auto G = microfacet_shadowing(a, n, h, o, i);
  return F * D * G / (4 * dot(n, o) * dot(n, i)) *
         dot(n, i);
}
```


© 2020 Fabio Pellacini

Microfacet reflection

- For micro facet reflections, our implementation follows the math formulation precisely

```
vec3f sample_specular(float ior, float a,
    vec3f n, vec3f o, vec2f rn) {
  if (dot(n, o) <= 0) return vec3f{0};
  auto h = sample_microfacet(a, n, rn);
  return reflect(o, h);
}

float sample_specular_pdf(float ior, float a,
    vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return 0;
  auto h = normalize(o + i);
  return sample_microfacet_pdf(a, n, h) /
         (4 * abs(dot(o, h)));
}
```


© 2020 Fabio Pellacini

Microfacet reflection

- Metals and dielectric reflections differ only on their fresnel term

```
vec3f eval_metal(vec3f eta, vec3f etak, float a,
    vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return vec3f{0};
  auto h = normalize(incoming + outgoing);
  auto F = fresnel_conductor(eta, etak, h, i);
  auto D = microfacet_distribution(a, n, h);
  auto G = microfacet_shadowing(a, n, h, o, i);
  return F * D * G / (4 * dot(n, o) * dot(n, i)) *
         dot(n, i);
}
vec3f sample_metal(vec3f eta, vec3f etak, float a,
    vec3f n, vec3f o, vec2f rn) {<same as sample_specular>}
float sample_specular_pdf(vec3f eta, vec3f etak, float a,
    vec3f n, vec3f o, vec3f i) {<sample_specular_pdf>}
```


© 2020 Fabio Pellacini

Microfacet reflection

- The micro facet functions are just an implementation of the formulae

```
float microfacet_distribution(float a, vec3f n, vec3f h) {
  auto c = dot(n, h); if (c <= 0) return 0;
  auto a2 = a * a; auto c2 = c * c;
  return a2 / (pi * (c2 * a2 + 1 - c2) * (c2 * a2 + 1 - c2));
}

float microfacet_shadowing1(float a, vec3f n,vec3f h, vec3f d) {
  auto c  = dot(n, d); auto ch = dot(h, d);
  if (c * ch <= 0) return 0;
  auto a2 = a * a; auto c2 = c * c;
  return 2 * abs(c) / (abs(c) + sqrt(c2 - a2 * c2 + a2));
}

float microfacet_shadowing(float a, vec3f n,
    vec3f h, vec3f o, vec3f i) {
  return microfacet_shadowing1(a, n, h, o) *
         microfacet_shadowing1(a, n, h, i);
}
```


© 2020 Fabio Pellacini

Microfacet reflection

- The microfacet functions are just an implementation of the formulae

```
vec3f sample_microfacet(float a, vec3f n, vec2f rn) {
  auto phi   = 2 * pi * rn.x;
  auto theta = atan(roughness * sqrt(rn.y / (1 - rn.y)));
  auto lh = vec3f{
      cos(phi) * sin(theta), sin(phi) * sin(theta), cos(theta)};
  return transform_direction(basis_fromz(n), lh);
}

float sample_microfacet_pdf(
    float a, vec3f n, const vec3f h) {
  auto c = dot(n, h); if (c < 0) return 0;
  return microfacet_distribution(a, n, h) * c;
}
```


© 2020 Fabio Pellacini

Generic BRDF

- As an example of a generic BRDF, we can write a BRDF as a weighted sum of predefined lobes; here we specify the lobes explicitly, but we could have also used an array of lobes

```
struct brdf {
  vec3f diffuse      = {0, 0, 0}; // diffuse weight
  vec3f specular     = {0, 0, 0}; // specular weight
  vec3f metal        = {0, 0, 0}; // metal weight
  vec3f transmission = {0, 0, 0}; // transmission weight
  float roughness    = 0;         // surface roughness
  float ior          = 1;         // specular ior
  vec3f meta         = {0, 0, 0}; // metal complex ior
  vec3f metak        = {0, 0, 0};
  float diffuse_pdf      = 0; // diffuse pdf
  float specular_pdf     = 0; // specular pdf
  float metal_pdf        = 0; // metal pdf
  float transmission_pdf = 0; // transmission pdf
};
```


© 2020 Fabio Pellacini

Generic BRDF

- The evaluation functions just sum each lobe contribution

```
vec3f eval_brdfcos(brdf brdf, vec3f n, vec3f o, vec3f i) {
 auto brdfcos = zero3f;
 if (brdf.diffuse)
   brdfcos += brdf.diffuse * eval_diffuse(n, o, i);
 if (brdf.specular)
   brdfcos += brdf.specular * eval_specular(brdf.ior,
                 brdf.roughness, n, o, i);
 if (brdf.metal)
   brdfcos += brdf.metal * eval_metal(brdf.meta, brdf.metak,
                 brdf.roughness, n, o, i);
 if (brdf.transmission)
   brdfcos += brdf.transmission * eval_transmission(brdf.ior,
                 brdf.roughness, n, o, i);
 return brdfcos;
}
```


© 2020 Fabio Pellacini

Generic BRDF

- The sample function picks a lobe by maintaining a running sum over the lobe pdfs

```
vec3f sample_brdfcos(brdf brdf, vec3f n, vec3f o,
  float rnl, vec2f rn) {
  auto cdf = 0.0f;
  if (brdf.diffuse_pdf) {
    cdf += brdf.diffuse_pdf;
    if (rnl < cdf) return sample_diffuse(n, o, rn); }
  if (brdf.specular_pdf) {
    cdf += brdf.specular_pdf;
    if (rnl < cdf) return sample_specular(brdf.r, n, o, rn); }
  if (brdf.metal_pdf) {
    cdf += brdf.metal_pdf;
    if (rnl < cdf) return sample_metal(brdf.r, n, o, rn); }
  if (brdf.transmission_pdf) {
    cdf += brdf.transmission_pdf;
    if (rnl < cdf)return sample_transmission(brdf.r, n, o, rn);}
  return zero3f;
}
```

Generic BRDF

- The sample pdf is computed as the sum of each lobe pdf

```
float sample_brdfcos_pdf(brdf brdf, vec3f n, vec3f o, vec3f i) {
  auto pdf = 0.0f;
  if (brdf.diffuse_pdf) {
    pdf += brdf.diffuse_pdf*sample_diffuse_pdf(n,o,i); }
  if (brdf.specular_pdf) {
    pdf += brdf.specular_pdf*sample_specular_pdf(brdf.r,n,o,i);}
  if (brdf.metal_pdf) {
    pdf += brdf.metal_pdf*sample_metal_pdf(brdf.r,n,o,i);}
  if (brdf.transmission_pdf) {
    pdf += brdf.transmission_pdf*
              sample_transmission_pdf(brdf.r,n,o,i);}
  return pdf;
}
```


© 2020 Fabio Pellacini

Area Formulation

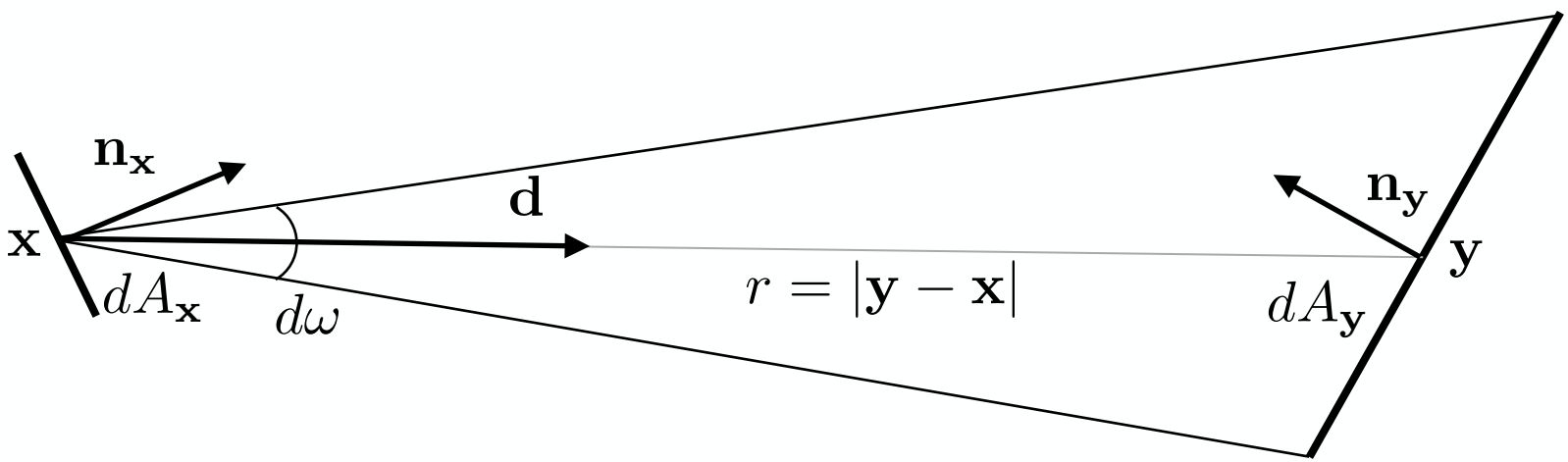
© 2020 Fabio Pellacini

Radiance between differential areas

- Radiance between surfaces can be computed considering the solid angle subtended by one surface w.r.t. the other

$$L(\mathbf{x}, \mathbf{d}_{\mathbf{x}\to\mathbf{y}}) = \frac{d^2P}{dA_{\mathbf{x}}(\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})d\omega_{\mathbf{d}}} = \frac{d^2P|\mathbf{y}-\mathbf{x}|^2}{dA_{\mathbf{x}}(\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})dA_{\mathbf{y}}(-\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})} =$$

$$\text{with } \mathbf{d}_{\mathbf{x}\to\mathbf{y}} = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|} \quad d\omega_{\mathbf{d}} = \frac{dA_{\mathbf{y}}(-\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})}{|\mathbf{y}-\mathbf{x}|^2}$$



© 2020 Fabio Pellacini

Rendering equation – area form

- Angular form of the rendering equation

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n}_{\mathbf{x}} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- Express the integral in terms of integrating over all possible surfaces in the scene; change the variable of integration from solid angle to area using the expression of the solid angle w.r.t. area
  - include a term *V* to decide whether two points are visible

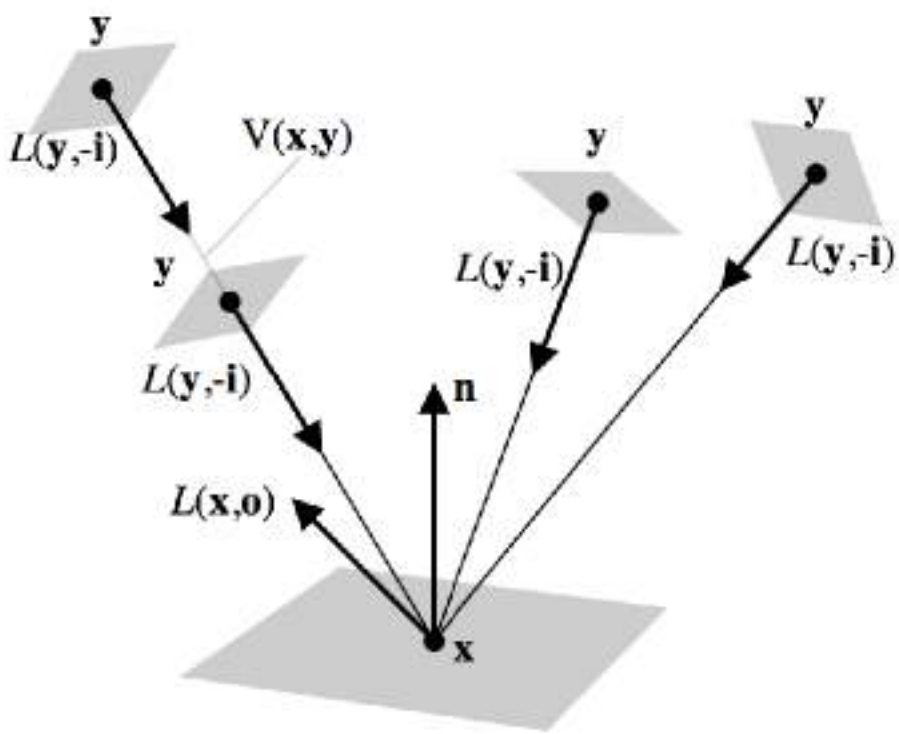
$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{\mathbf{y}\in S} L(\mathbf{y},-\mathbf{i}) V(\mathbf{x},\mathbf{y}) f(\mathbf{x},\mathbf{i},\mathbf{o}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$

$$\text{with } \mathbf{i} = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|} \quad d\omega_{\mathbf{i}} = \frac{dA_{\mathbf{y}} |-\mathbf{i}\cdot\mathbf{n}_{\mathbf{y}}|}{|\mathbf{y}-\mathbf{x}|^2} \quad G(\mathbf{x},\mathbf{y}) = \frac{|\mathbf{i}\cdot\mathbf{n}_{\mathbf{x}}||-\mathbf{i}\cdot\mathbf{n}_{\mathbf{y}}|}{|\mathbf{y}-\mathbf{x}|^2}$$

© 2020 Fabio Pellacini

Rendering equation – area form

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{\mathbf{y}\in S} L(\mathbf{y},-\mathbf{i})V(\mathbf{x},\mathbf{y})f(\mathbf{x},\mathbf{i},\mathbf{o})G(\mathbf{x},\mathbf{y})dA_{\mathbf{y}}$$

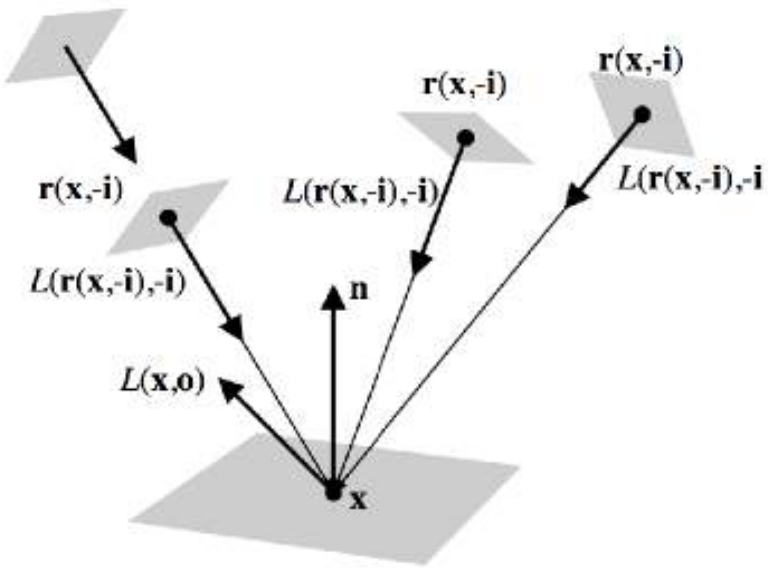


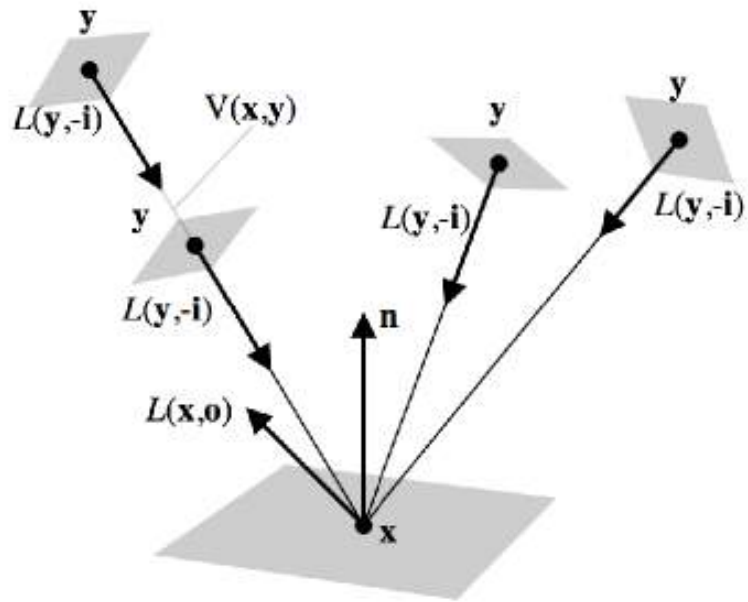
© 2020 Fabio Pellacini

Angular vs area form

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_{\mathbf{i}}$$

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{\mathbf{y}\in S} L(\mathbf{y},-\mathbf{i}) V(\mathbf{x},\mathbf{y}) f(\mathbf{x},\mathbf{i},\mathbf{o}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$





© 2020 Fabio Pellacini

Angular vs area form

- **Angular form**: integrate over incoming angle
  - use raytracing to determine the visible surface from where the light comes from
- **Area form**: integrate over all scene's surfaces
  - use raytracing to determine whether a given point is visible from another, i.e. whether a point contributes to the integral (shadow rays)

© 2020 Fabio Pellacini

Direct illumination – area

- Consider the same solution but now with the area formulation

$$L_d(\mathbf{x}, \mathbf{o}) = \int_{\mathbf{y} \in A_{light}} L_e(\mathbf{y}, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}) G(\mathbf{x}, \mathbf{y}) dA_{\mathbf{y}}$$

- The estimator for this formulation is

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{y}_i, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}_i) G(\mathbf{x}, \mathbf{y}_i)}{p(\mathbf{y}_i)}$$

- To evaluate it, we have to pick points uniformly over the light surface

© 2020 Fabio Pellacini

Sampling multiple lights

- To handle multiple lights, we can estimate their contributions independently since each light integral is independent

$$L_d(\mathbf{x},\mathbf{o}) = \int_{\mathbf{y}\in A_{lights}} L_e(\mathbf{y},-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) V(\mathbf{x},\mathbf{y}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}} =$$

$$= \sum_l^{n_l} \int_{\mathbf{y}\in A_l} L_e(\mathbf{y},-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) V(\mathbf{x},\mathbf{y}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$

$$L_d(\mathbf{x},\mathbf{o}) \approx \sum_l^{n_l} \frac{A_{l_i}}{N} \sum_i A_{l_i} L_e(\mathbf{y}_i,-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) V(\mathbf{x},\mathbf{y}_i) G(\mathbf{x},\mathbf{y}_i)$$

© 2020 Fabio Pellacini

Sampling multiple lights

- As the number of lights increases, this formulation becomes inefficient
- The alternative is to pick a *light at random for each sample*
- Here we are sampling from a uniform *discreet* distribution

$$L_d(\mathbf{x}, \mathbf{o}) = \sum_l^{n_l} \int_{\mathbf{y} \in A_l} L_e(\mathbf{y}, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}) G(\mathbf{x}, \mathbf{y}) dA_{\mathbf{y}}$$

$$L_d(\mathbf{x}, \mathbf{o}) \approx A_{l_i} L_e(\mathbf{y}_i, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}_i) G(\mathbf{x}, \mathbf{y}_i)$$

$$l_i = \mathrm{floor}(r \cdot n_l) \quad l \sim p(l) = \frac{1}{n_l}$$

© 2020 Fabio Pellacini

Proof of Recursive Solution

© 2020 Fabio Pellacini

Rendering eq. - operator form

- We can think of reflection as transport operator *T* that takes as input the radiance function and return a new radiance function

$$T : L(\mathbf{x}, \mathbf{o}) \rightarrow L_r(\mathbf{x}, \mathbf{o})$$

$$(TL)(\mathbf{x}, \mathbf{o}) = \int_{\mathbf{y} \in S} L(\mathbf{y}, -\mathbf{i}) V(\mathbf{x}, \mathbf{y}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) G(\mathbf{x}, \mathbf{y}) dA_{\mathbf{y}}$$

- Rendering equation

$$L(\mathbf{x}, \mathbf{o}) = L_e(\mathbf{x}, \mathbf{o}) + (TL)(\mathbf{x}, \mathbf{o})$$

- In shorthand notation

$$L = L_e + TL$$

© 2020 Fabio Pellacini

Rendering eq. - operator form

- Formal solution of the rendering equation is

$$L = L_e + TL \rightarrow (I - T)L = L_e \rightarrow L = (I - T)^{-1} L_e$$

- Unusable in practice since we cannot compute the inverse
- Solve by recursive substitution

$$L = L_e + TL = L_e + T(L_e + TL) = L_e + TL_e + T^2 L =$$
$$= L_e + TL_e + T^2(L_e + TL) = L_e + TL_e + T^2 L_e + T^3 L = \ldots$$

- This leads to the solution written as a Neumann series

$$L = \sum_{i=0}^{\infty} T^i L_e$$

© 2020 Fabio Pellacini

Rendering eq. - operator form

- We can show that the Neumann series is the right solution by equating it in the formal solution

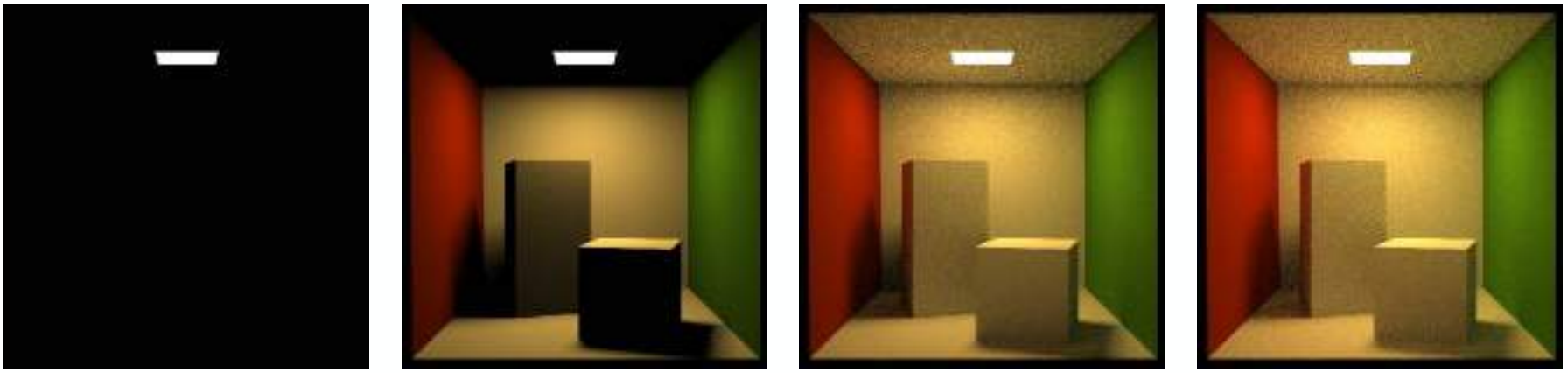
$$L = \sum_{i=0}^{\infty} T^i L_e \qquad L = (I - T)^{-1} L_e$$

- Since these equations are true for every *Le*, we have that

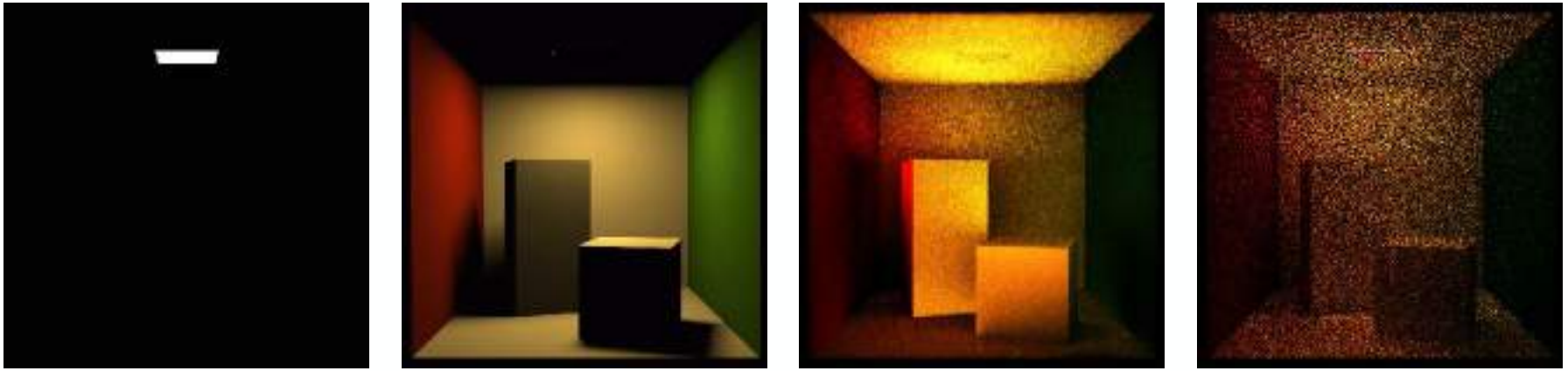
$$\begin{aligned}
(I - T)^{-1} &= \sum_{i=0}^{\infty} T^i L_e \\
(I - T)(I - T)^{-1} &= (I - T) \sum_{I=0}^{\infty} T^i L_e \\
I &= \sum_{i=0}^{\infty} T^i L_e - \sum_{i=1}^{\infty} T^i L_e \\
I &= I
\end{aligned}$$

© 2020 Fabio Pellacini

Rendering eq. - operator form



L_e $\quad L_e + TL_e \quad L_e + TL_e + T^2 L_e \quad L_e + ... + T^3 L_e$



$L_e \quad TL_e \quad T^2 L_e$ (16x) $\quad T^3 L_e$ (16x)

[Dutrè, Bala]

© 2020 Fabio Pellacini

Properties of TL

- Intuition: *TL* bounces light once
- From the conservation of energy, we know that the norm of *TL* will be less or equal to the one of *L*
  - since light is either reflected of absorbed
  - similar to energy conservation for the BSDF
- The higher the BSDFs of the scene the more bounces we need since energy decreases less at each bounce

© 2020 Fabio Pellacini

Recursive solution

- We know now how to recursively solve the rendering equation

$$L = \sum_{i=0}^{\infty} T^i L_e$$

- Let us rewrite this in terms of the original integrals

$$\begin{aligned} L(\mathbf{x}_0, \mathbf{o}_0) = L_e(\mathbf{x}_0, \mathbf{o}_0) + \\ + \int_{\mathbf{x}_1} L_e(\mathbf{x}_1, \mathbf{o}_1) V(\mathbf{x}_0, \mathbf{x}_1) f(\mathbf{x}_0, \mathbf{i}_0, \mathbf{o}_0) G(\mathbf{x}_0, \mathbf{x}_1) dA_{\mathbf{x}_1} + \\ + \int_{\mathbf{x}_1} \left(\int_{\mathbf{x}_2} L_e(\mathbf{x}_2, \mathbf{o}_2) V(\mathbf{x}_1, \mathbf{x}_2) f(\mathbf{x}_1, \mathbf{i}_1, \mathbf{o}_1) G(\mathbf{x}_1, \mathbf{x}_2) dA_{\mathbf{x}_2} \right) \cdot \\ \cdot V(\mathbf{x}_0, \mathbf{x}_1) f(\mathbf{x}_0, \mathbf{i}_0, \mathbf{o}_0) G(\mathbf{x}_1, \mathbf{x}_2) dA_{\mathbf{x}_2} + \ldots \end{aligned}$$

© 2020 Fabio Pellacini

Recursive solution

- We can write the recursive solution by aggregating the product of all visibility, BRDF and geometric terms in a single expression called *path throughput*

$$\mathrm{Tr}(\mathbf{x}_0, ..., \mathbf{x}_i) = \prod_{j=0}^{i-1} V(\mathbf{x}_j, \mathbf{x}_j + 1) f(\mathbf{x}_j, \mathbf{i}_j, \mathbf{o}_j) G(\mathbf{x}_j, \mathbf{x}_{j+1})$$

$$L(\mathbf{x}, \mathbf{o}) = \sum_{i=0}^{\infty} \int_{\mathbf{x}_1} ... \int_{\mathbf{x}_i} L_e(\mathbf{x}_i, \mathbf{o}_i) \mathrm{Tr}(\mathbf{x}_0, ..., \mathbf{x}_i) dA_{\mathbf{x}_1} ... dA_{\mathbf{x}_i}$$

© 2020 Fabio Pellacini

Rendering eq. – path integral form

- From the previous formulation, we can get some intuition on yet another form of the rendering equation called *path formulation*
- The basic idea is to think about integrating not recursively over position but in *space of all possible paths* that the light can travel into
- A light path is a sequence of points of *any length*

$$\mathbf{p} = \{\mathbf{x}_0, ..., \mathbf{x}_i\}$$

- We can rewrite the previous solution as the integral of the radiance of all possible path that end at the camera

$$L(\mathbf{x}, \mathbf{o}) = \int_{\mathbf{p} \in \mathcal{P}} L_e(\mathbf{p}_{end}) \mathrm{Tr}(\mathbf{p}) d\mu_{\mathbf{p}}$$

© 2020 Fabio Pellacini

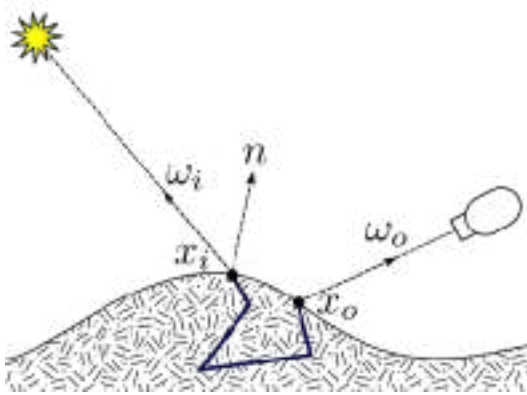
Subsurface Scattering

© 2020 Fabio Pellacini

Subsurface scattering: BSSRDF

- BSSRDF [$m^{-2}sr^{-1}$]: *bidirectional scattering surface-reflectance distr. func*.
- Defined as the ratio of the differential reflected radiance over the differential incident flux
- Depends on incoming and outgoing directions, incoming and outgoing positions, and wavelength of light
- Main different w.r.t. BSDF: photons leave at different points then they enter, so consider two both locations in the equation

$$s(\mathbf{x_i}, \mathbf{x_o}, \mathbf{i}, \mathbf{o}) = \frac{dL_r(\mathbf{x_o}, \mathbf{o})}{dP_i(\mathbf{x_i}, \mathbf{i})} = \frac{dL_r(\mathbf{x_o}, \mathbf{o})}{L_i(\mathbf{x_i}, \mathbf{i})|\mathbf{i} \cdot \mathbf{n_i}| d\omega_\mathbf{i} dA_{\mathbf{x_i}}}$$



© 2020 Fabio Pellacini

Subsurface scattering: BSSRDF

- Reflected radiance can be obtained by integrating the above equation over all points on the surface and for each incoming direction around each point

$$L_r(\mathbf{x_o}, \mathbf{o}) = \int_{\mathbf{x_i} \in S} \int_{\mathbf{i} \in H^2} L_i(\mathbf{x_i}, \mathbf{i}) s(\mathbf{x_i}, \mathbf{x_o}, \mathbf{i}, \mathbf{o}) |\mathbf{i} \cdot \mathbf{n_i}| d\omega_{\mathbf{i}} dA_{\mathbf{x_i}}$$

© 2020 Fabio Pellacini

Properties of the BSSRDF

- Same properties of BSSRDF
- Positivity: since photons cannot be "negatively" reflected

$$s(\mathbf{y}, \mathbf{x}, \mathbf{i}, \mathbf{o}) \geq 0$$

- Helmotz reciprocity: can invert light paths

$$s(\mathbf{y}, \mathbf{x}, \mathbf{i}, \mathbf{o}) = s(\mathbf{x}, \mathbf{y}, \mathbf{o}, \mathbf{i})$$

- Energy conservation: all photons that comes in are either reflected or absorbed and no new photons is emitted

$$\int_S \int_{H^2} s(\mathbf{y}, \mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{i} \cdot \mathbf{o}| d\omega_{\mathbf{i}} dA_{\mathbf{y}} \leq 1$$

© 2020 Fabio Pellacini

Rendering with BSSRDF

BRDF

BSSRDF

[Jensen et al]

© 2020 Fabio Pellacini

Rendering with BSSRDF

[Conner et al]

[Jensen et al]

© 2020 Fabio Pellacini

Volumetric Rendering

© 2020 Fabio Pellacini

Volume scattering processes

- Emission: photons are emitted by the medium

[https://wikipedia.org]

© 2020 Fabio Pellacini

Volume scattering processes

- Absorption: photons are absorbed by the medium

[https://common.wikipedia.org]

© 2020 Fabio Pellacini

Volume scattering processes

- Scattering: photons change direction in the medium

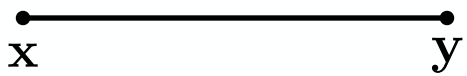
[https://coclouds.com]

© 2020 Fabio Pellacini

Volume scattering processes

- Surface-only scattering: radiance is constant along a ray

$$L_i(\mathbf{x}, \mathbf{i}) = L_o(\mathbf{y}, -\mathbf{i})$$



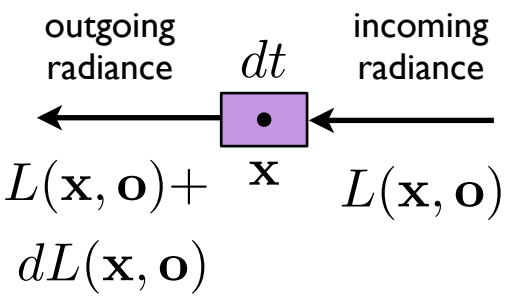
- Volume scattering: describe how radiance changes along a ray

$$L_i(\mathbf{x}, \mathbf{i}) \neq L_o(\mathbf{y}, -\mathbf{i})$$

© 2020 Fabio Pellacini

Volume scattering processes

- Main idea: describe how radiance changes on a differential beam of length *dt* and area *dA* at a point *x* and oriented in direction *o*

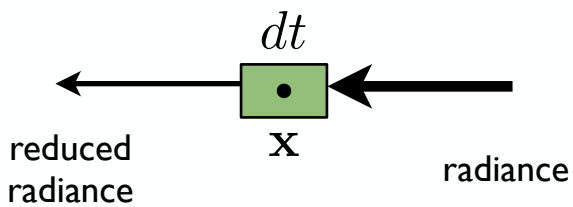


© 2020 Fabio Pellacini

Volume scattering processes

- Absorption: reduction of radiance due to absorption of radiance
- Absorption coefficient [m^{-1}]: describes the absorbed radiance per unit length

$$dL(\mathbf{x}, \mathbf{o}) = -\sigma_a(\mathbf{x}) L(\mathbf{x}, \mathbf{o}) dt$$

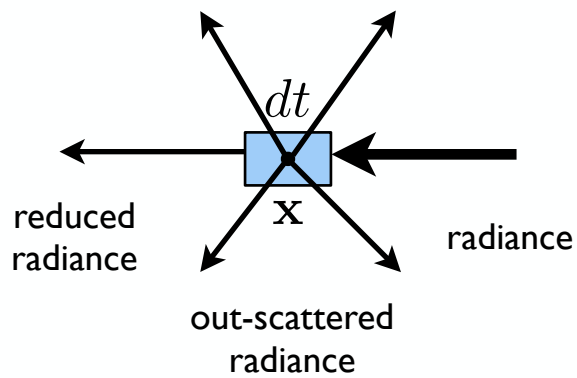


© 2020 Fabio Pellacini

Volume scattering processes

- Out scattering: reduction of radiance due to scattering in other directions
- Scattering coefficient [m^{-1}]: describes the scattered radiance per unit length

$$dL(\mathbf{x}, \mathbf{o}) = -\sigma_s(\mathbf{x}) L(\mathbf{x}, \mathbf{o}) dt$$

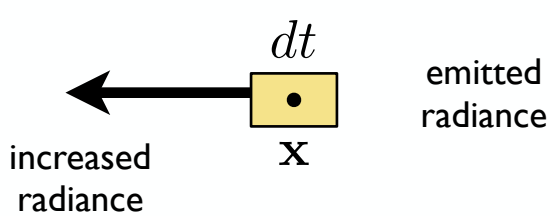


© 2020 Fabio Pellacini

Volume scattering processes

- Emission: increase of radiance due to emission of radiance
- Absorption coefficient [m^{-1}]: describes the absorbed radiance per unit length
- Emitted radiance as before

$$dL(\mathbf{x}, \mathbf{o}) = \sigma_a(\mathbf{x}) L_e(\mathbf{x}, \mathbf{o}) dt$$

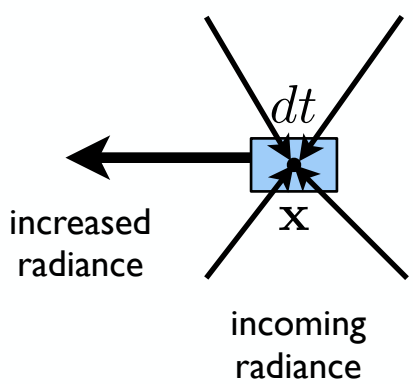


© 2020 Fabio Pellacini

Volume scattering processes

- In scattering: increase of radiance due to scattering from other directions
- Scattering coefficient [m^{-1}]: describes the scattered radiance per unit length

$$dL(\mathbf{x}, \mathbf{o}) = \sigma_s(\mathbf{x}) L_s(\mathbf{x}, \mathbf{o}) dt$$

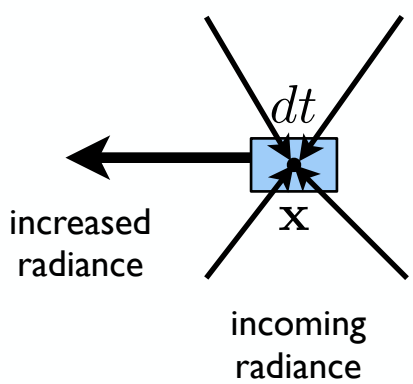


© 2020 Fabio Pellacini

Volume scattering processes

- In-scattered radiance: radiance scattered from all directions into this one, computed as the integral over the sphere of the incoming radiance weighted by the phase function
- Phase function: describes the angular distribution of scattering

$$L_s(\mathbf{x}, \mathbf{o}) = \int_S f_p(\mathbf{x}, \mathbf{o}, \mathbf{i}) L_i(\mathbf{x}, \mathbf{i}) d\omega_{\mathbf{i}}$$

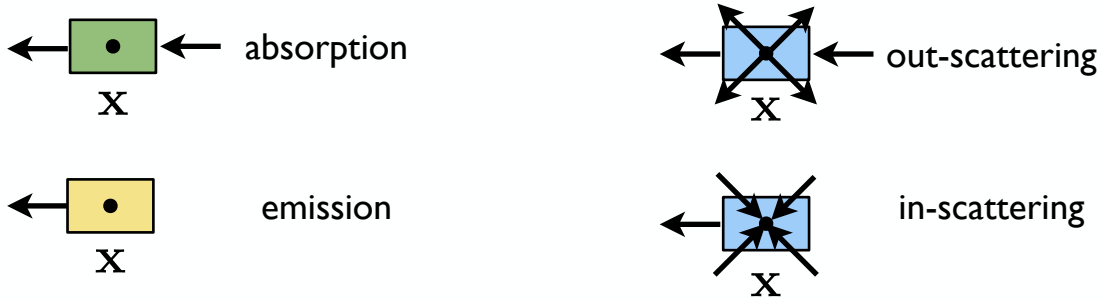


© 2020 Fabio Pellacini

Volume scattering processes

- In summary, at a point *x*, radiance decreases due to absorption and out-scattering and increases due to emission and in-scattering

$$
\begin{aligned}
dL(\mathbf{x},\mathbf{o}) = &-\sigma_a(\mathbf{x})L(\mathbf{x},\mathbf{o})dt \;\; -\sigma_s(\mathbf{x})L(\mathbf{x},\mathbf{o})dt \\
&+\sigma_a(\mathbf{x})L_e(\mathbf{x},\mathbf{o})dt + \sigma_s(\mathbf{x})L_s(\mathbf{x},\mathbf{o})dt
\end{aligned}
$$



© 2020 Fabio Pellacini

Volume scattering processes

- For convenience, we can parametrize the previous equation
- Extinction coefficient [m^{-1}]: measures the total decrease in radiance per unit length; defined as the sum of absorption and scattering
- Albedo [unit]: measures the amount of scattered radiance independently of extinction; defined as ratio of scattering and extinction

$$dL(\mathbf{x},\mathbf{o}) = -\sigma_t(\mathbf{x})L(\mathbf{x},\mathbf{o})dt + \sigma_a(\mathbf{x})L_e(\mathbf{x},\mathbf{o})dt + \sigma_s(\mathbf{x})L_s(\mathbf{x},\mathbf{o})dt$$

$$\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x}) \qquad \rho_s(\mathbf{x}) = \sigma_s(\mathbf{x})/\sigma_t(\mathbf{x})$$

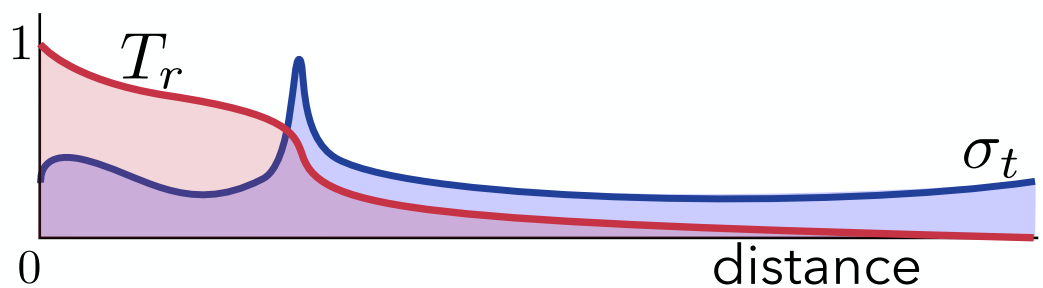
© 2020 Fabio Pellacini

Volumetric transmittance

- Volumetric transmittance [unit]: decrease in radiance between 2 points
- Transmittance is the points radiances and is between 0 and 1
- We can compute by integrating the excitation terms along a ray

$$T(\mathbf{x}, \mathbf{x}_t) = e^{-\int_0^t \sigma_t(\mathbf{x}_s) ds}$$

$$\mathbf{x}_t = \mathbf{x} - t\mathbf{o}$$



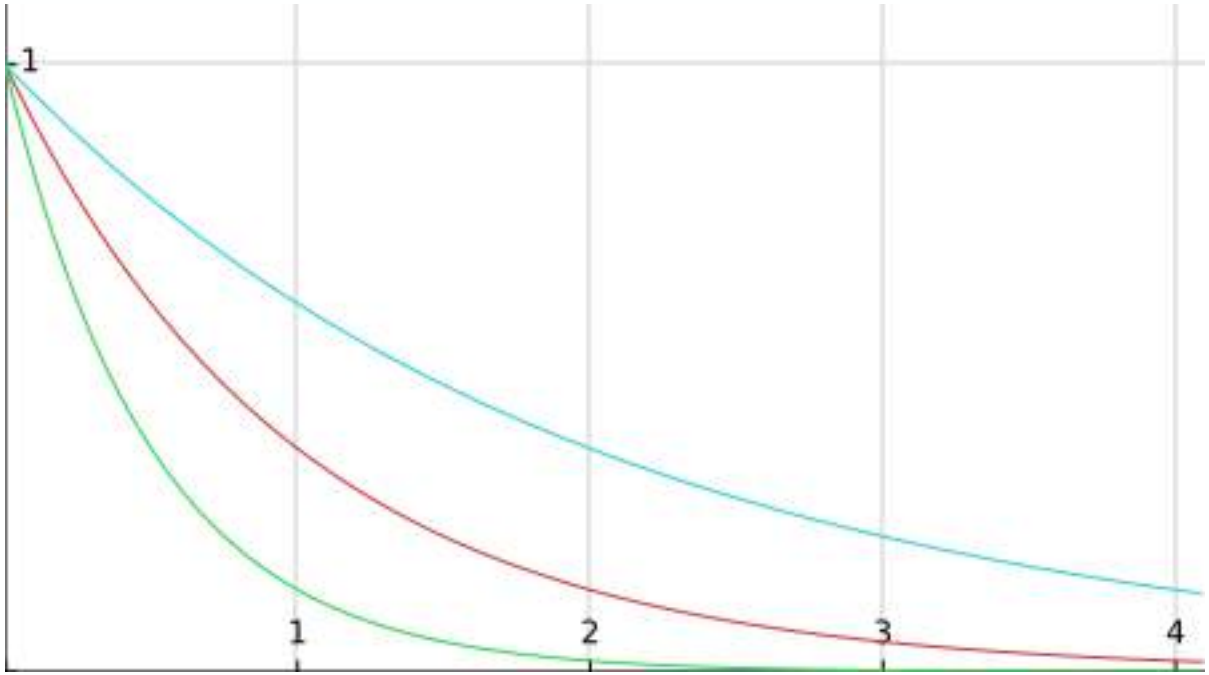
[Jaroz]

© 2020 Fabio Pellacini

Volumetric transmittance

- For homogenous media, that has a constance extinction coefficient, we get the know simpler Beer's law

$$T(\mathbf{x}, \mathbf{x}_t) = e^{-\sigma_t t}$$



© 2020 Fabio Pellacini

Volumetric rendering equation

- We can now derive the volumetric rendering equation by integrating along a ray from a point to another
- The derivation is non-trivial and hardly required to be able to solve the equation itself
- So we will skip the derivation and focus on under the equation itself
- To make integration easier in a path tracer, we are going to use the direction conventions introduce for surface scattering

© 2020 Fabio Pellacini

Volumetric rendering equation

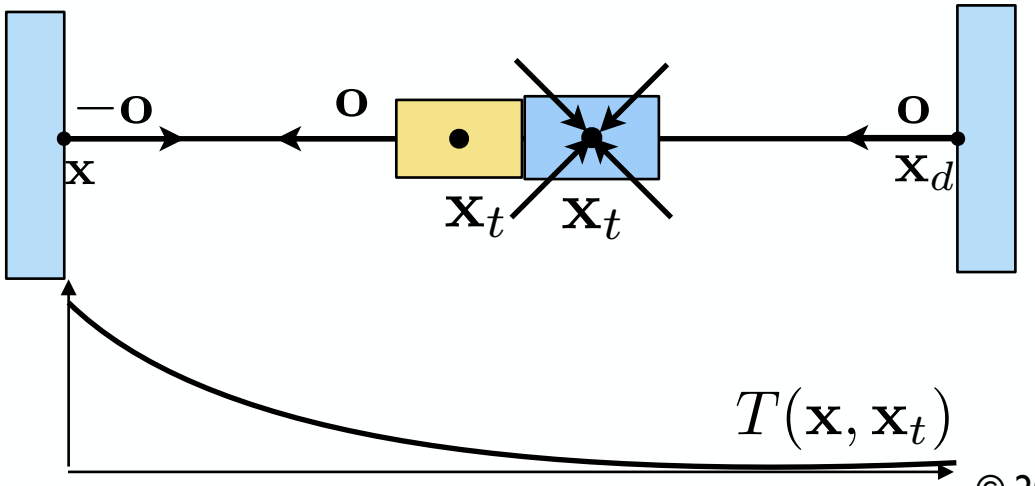
- The volumetric rendering equation is the sum of three terms
  - Background radiance rescaled by transmittance
  - Emitted radiance rescaled and accumulated along the ray
  - In-scattered radiance rescaled and accumulated along the ray

$$
\begin{aligned}
L_i(\mathbf{x}, -\mathbf{o}) = T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o}) \\
+ \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) dt \\
+ \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o}) dt
\end{aligned}
$$

© 2020 Fabio Pellacini

Volumetric rendering equation

$$
\begin{aligned}
L_i(\mathbf{x}, -\mathbf{o}) = T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o}) \\
+ \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) dt \\
+ \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o}) dt
\end{aligned}
$$



© 2020 Fabio Pellacini

Volumetric path tracing

- We want to extend our path tracer to integrate volumes
- There are several manner to do this, with different tradeoffs
- In our case, we want to extend a path along one direction only, as before, to avoid exponential growth
- For simplicity, we will model volumes as enclosed into surfaces that are either the surface of real-world objects, like glass, as fictitious surfaces to bound the volume, like a fog proxy volume
- In this case, a path can either
  - reflect/refract on a surface
  - scatter inside the medium
- Once again, the resulting Monte Carlo algorithm is simpler than the math that describes it

© 2020 Fabio Pellacini

Volumetric path tracing

- Basic idea: split the integral into surface and volume contributions

surface
contribution

$$L_i(\mathbf{x}, -\mathbf{o}) = T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o})$$
$$+ \int_0^d T(\mathbf{x}, \mathbf{x}_t)[\sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) + \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o})] dt$$

volume
contribution

© 2020 Fabio Pellacini

Volumetric path tracing

- To estimate it, sample a distance along a path and either estimate the surface contribution or the volume one
- Now we have *Le*, but need to estimate *Ls*

surface estimate
using path tracing

$$L_i(\mathbf{x}, -\mathbf{o}) \approx \frac{T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o})}{P(d)}$$

probability of
picking surface
at distance d

$$+ \frac{T(\mathbf{x}, \mathbf{x}_t)[\sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) + \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o})]}{p(t)}$$

pdf for sampling
distance t

volume estimate
by scattering in volume

© 2020 Fabio Pellacini

Volumetric path tracing

- To estimate the in-scattered radiance, pick one direction according to the phase function and continue the path; can also integrate with MIS

$$L_i(\mathbf{x}, -\mathbf{o}) \approx \frac{T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o})}{P(d)} + \frac{T(\mathbf{x}, \mathbf{x}_t)[\sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) + \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o})]}{p(t)}$$

$$L_s(\mathbf{x}_t, \mathbf{o}) \approx \frac{f_p(\mathbf{x}_t, \mathbf{i}, \mathbf{o}) L_i(\mathbf{x}_t, \mathbf{i})}{p(\mathbf{i})}$$

pdf for sampling direction i

© 2020 Fabio Pellacini

Volumetric path tracing

- We can write a volumetric path tracer starting from a standard one
- Here we consider the simplification of having only smooth birds

```
vec3f shade_path(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    <emission>
    <sample brdf direction>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


© 2020 Fabio Pellacini

Volumetric path tracing

- Volumes are contained into surfaces that are used to delimit them
- We send a ray to compute the maximum distance to the surface
- After intersection, we sample a distance and either scatter in the volume or on the surface

```
vec3f shade_volpath(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <sample transmittance>
    if(!in_volume) <handle surface>
    else <handle volume>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


© 2020 Fabio Pellacini

Volumetric path tracing

- We track the objects we are in with a stack
- We then sample the free-flight distance and update the weight by the transmission over its pdf; for simplicity here we handle both surface and volume interactions

```
<sample transmittance>
  auto in_volume = false;
  if (!vstack.empty()) {
    auto extinction = volume_stack.back().extinction;
    auto distance = sample_transmittance(
          extinction, isec.distance, rand1f(rng), rand1f(rng));
    W *= eval_transmittance(extinction, distance) /
         sample_transmittance_pdf(extinction, distance,
                                  isec.distance);
    in_volume = distance < isec.distance;
    isec.distance = distance;
  }
```


© 2020 Fabio Pellacini

Volumetric path tracing

- The surface interaction is the same as a standard path tracer, except that we update the volume stack upon entering and exiting objects.

```
<handle surface> {
  auto [p, n] = eval_point(isec);     // eval pos, norm
  auto [e, f] = eval_material(isec);  // eval bsdf
  auto o = -ray.d;                    // outgoing
  l += w * eval_emission(e, n, o);    // emission
  auto i = rand1f(rng) < 0.5 ?        // incoming
     sample_brdfcos(n, o, rand1f(rng),rand2f(rng)) :
     sample_lights(p, rand1f(rng),rand1f(rng),rand2f(rng));
  w *= eval_brdfcos(f,n,i,o) * 2 /
    (sample_brdfcos_pdf(f,n,i,o) + sample_lights_pdf(p,i));
  ray = {p, i};                       // setup recurse
  if (has_volume(isec) && dot(n,o)*dot(n,i)<0){// if change side
    if (volume_stack.empty())         // update volume stack
      vstack.push_back(eval_volume(isec));
    else vstack.pop_back();
}
```

Volumetric path tracing

- The volume interaction is similar to the surface one, but we scatter with respect to the phase function rather than the bsdf

```
<handle volume> {
  auto p = eval_ray(ray, isec.distance); // eval pos
  auto vol = vstack.back();              // get volume props
  auto o = -ray.d;                     // outgoing
  l += w * eval_emission(vol, p, o);  // emission
  auto i = rand1f(rng) < 0.5 ?        // incoming
     sample_scattering(vol,o,rand2f(rng)) :
     sample_lights(p,rand1f(rng),rand1f(rng),rand2f(rng));
  w *= eval_scattering(vol,i,o) * 2 /
    (sample_scattering_pdf(vol,i,o) + sample_lights_pdf(p,i));
  ray = {p, i};                        // setup recurse
}
```


© 2020 Fabio Pellacini

Volumetric path tracing

- To write our estimator, we need to write functions to
  - Sample a distance in the medium, and compute the distance pdf
  - Compute the transmission at a given distance
  - Sample the phase function and compute its PDF
- We'll now introduce each of these pieces

© 2020 Fabio Pellacini

Homogeneous volumes

- Handling distance sampling and computing transmittance is easy for homogenous volumes, i.e. volumes that have constant parameters, since analytic solutions are available for all computations
- Homogenous volumes are particularly important since they can be used to model skin and refractive glass.

© 2020 Fabio Pellacini

Homogeneous volumes

- For homogeneous volumes, transmittance is just Beer's law

$$T(t) = e^{-\sigma_t t}$$

- We want to sample distances with a PDF that is proportional to transmittance; this makes for an efficient algorithm since volumes with low transmittance will more likely sample the surface and vice-versa
- As before we use a warp for which we do not derive the formula

$$t = -\frac{\ln(1-r)}{\sigma_t} \qquad p(t) = \sigma_t e^{-\sigma_t t}$$

- From this pdf, we can compute the probability of hitting the surface

$$P(d) = \int_d^\infty p(t)dt = e^{-\sigma_t d}$$

© 2020 Fabio Pellacini

Homogeneous volumes

- For colored extinction, we cannot directly use the previous formulae, since sampling distance depends on wavelength
- In this case, we can simple apply MIS over the channels
- The sampling function first pick a channel and then samples according to that, while the pdf is computed as the weighted sum of pdfs

© 2020 Fabio Pellacini

Homogeneous volumes

- The transmittance functions are straightforward implementations

```
float sample_transmittance(
    vec3f extinction, float max_distance, float rl, float rd) {
  auto channel  = clamp((int)(rl * 3), 0, 2);
  auto distance = (extinction[channel] == 0) ?
        float_max : -log(1 - rd) / extinction[channel];
  return min(distance, max_distance);
}
float sample_transmittance_pdf(
    vec3f& extinction, float distance, float max_distance) {
  if(distance < max_distance) {
    return sum(extinction * exp(-extinction*distance)) / 3;
  } else {
    return sum(exp(-extinction*max_distance)) / 3;
  }
}
vec3f eval_transmittance(vec3f dextinction, float distance) {
  return exp(-extinction * distance);
}
```


© 2020 Fabio Pellacini

Phase function

- Directional scattering is controlled by the phase function
- As before, many model exists to capture different behaviors
- Phase functions are the equivalent of the BSDF for volumes
- The simplest phase function is just a constant, indicating that light is scattered in every direction equally

$$f_p(\mathbf{x}, \mathbf{i}, \mathbf{o}) = \frac{1}{4\pi}$$

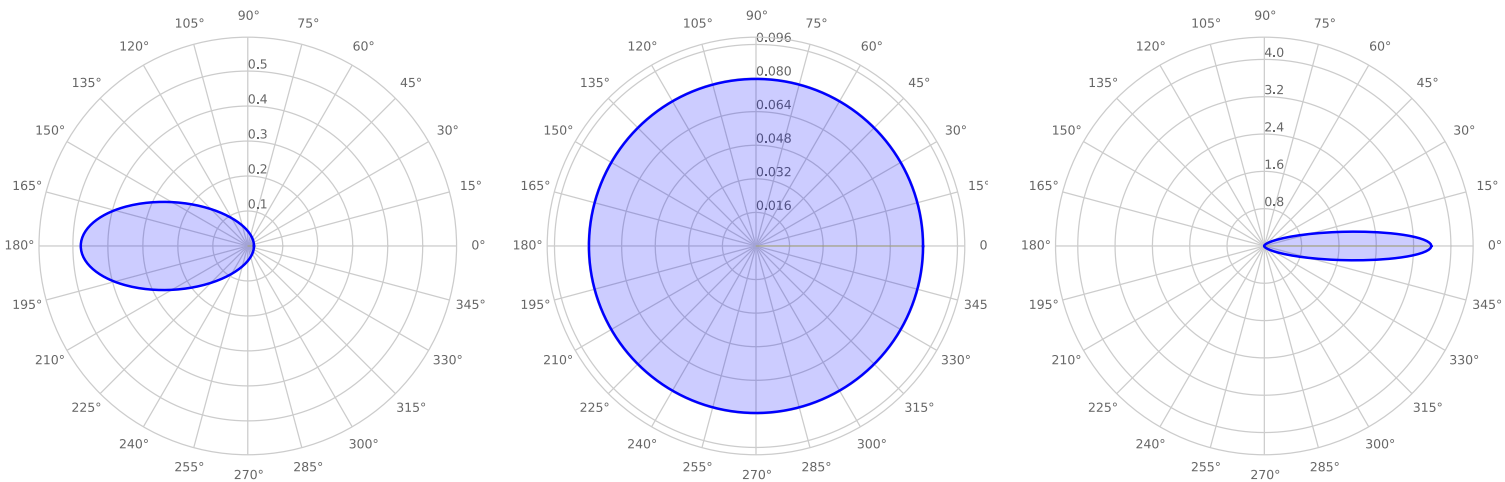
© 2020 Fabio Pellacini

Phase function

- The most commonly used phase function is the Henyey-Greenstein
- It can describe isotropic materials, but also anisotropic materials, that scatter mostly into one direction

$$f_p(\mathbf{i}, \mathbf{o}) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g(\mathbf{i} \cdot \mathbf{o}))^{3/2}}$$

$g = -0.5$ $\quad$ $g = 0$ $\quad$ $g = 0.8$



[Jaroz]

© 2020 Fabio Pellacini

Phase function

- For the Henyey-Greenstein phase functions, we can use a warp to pick directions with a pdf proportional to the phase function itself

$$\cos\theta = \frac{1}{2g}\left(1 + g^2 - (\frac{1-g^2}{1-g+2gr_2})^2\right)$$

$$\phi = 2\pi r_2$$

$$p(\mathbf{i}, \mathbf{o}) = f_p(\mathbf{i}, \mathbf{o})$$

- We need special handling for g=0

$$\cos\theta = 1 - 2r_2$$

© 2020 Fabio Pellacini

Phase function

- The volume functions are just wrappers over simpler functions

```
vec3f eval_emission(vsdf vol, vec3f p, vec3f o) {
  return vol.extinction * (1 - vol.albedo) * vol.emission;
}

vec3f eval_scattering(vsdf vol, vec3f i, vec3f o) {
  return vol.extinction * vol.albedo *
    eval_phasefunc(vol.anisotropy, i, o);
}

vec3f sample_scattering(vsdf vol, vec3f i, vec3f o) {
  return sample_phasefunc(vol.anisotropy, o);
}

vec3f pdf_scattering(vsdf vol, vec3f i, vec3f o) {
  return pdf_phasefunc(vol.anisotropy, i, o);
}
```


© 2020 Fabio Pellacini

Phase function

- The scattering function functions are straightforward implementation

```
vec3f eval_phasefunc(float g, vec3f i, vec3f o) {
  auto c = -dot(i, o);
  return (1 - g*g) / (4 * pi * pow(1 + g*g - 2*g*c,3/4));
}
vec3f sample_phasefunc(float g, vec3f o, vec2f rn) {
  auto cos_theta = 0.0f;
  if (abs(g) < 1e-3f) { cos_theta = 1 - 2 * rn.y; } else {
    auto square = (1 - g * g) / (1 - g + 2 * g * rn.y);
    cos_theta    = (1 + g * g - square * square) / (2 * g);
  }
  auto sin_theta = sqrt(max(0, 1 - cos_theta * cos_theta));
  auto phi       = 2 * pif * rn.x;
  auto il = {sin_theta*cos(phi), sin_theta*sin(phi), cos_theta};
  return basis_fromz(-o) * il;
}
vec3f sample_phasefunc_pdf(float g, vec3f i, vec3f o) {
  return eval_phasefunc(vol.anisotropy, i, o);
}
```


© 2020 Fabio Pellacini

Smooth and Rough Surfaces

Prof. Fabio Pellacini

© 2020 Fabio Pellacini •

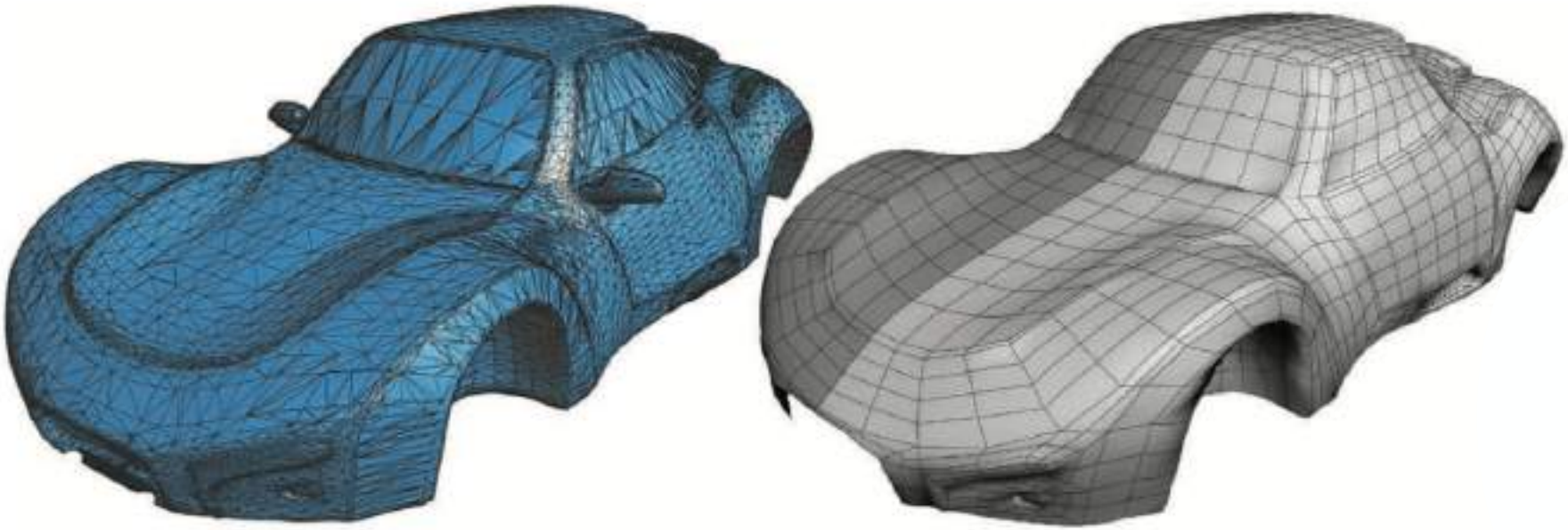
Triangle mesh recap

- Triangles have well defined geometric properties
  - they always live in a plane
  - so they have a unique geometric normal
- Triangle meshes are widely used
  - supported in rasterization and raytracing
  - sometimes used in simulation for cloth
- But it is hard to represent "regular" structures with triangles

© 2020 Fabio Pellacini •

Quad meshes

- Quad mesh: a collection of quadrilaterals
  - 4 vertices per face, possibly non-planar
  - use same data structure that triangles use

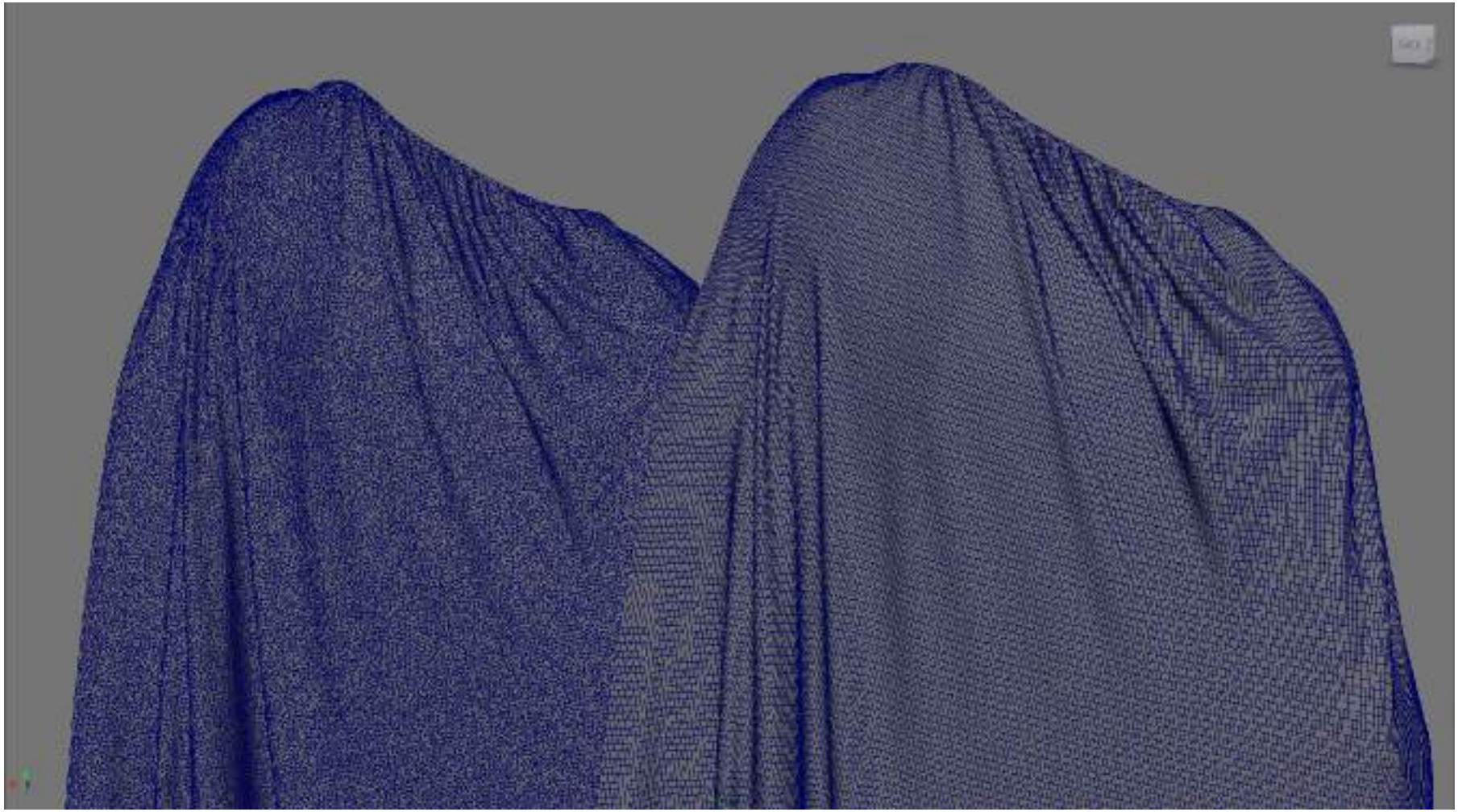


[Panozzo]

© 2020 Fabio Pellacini •

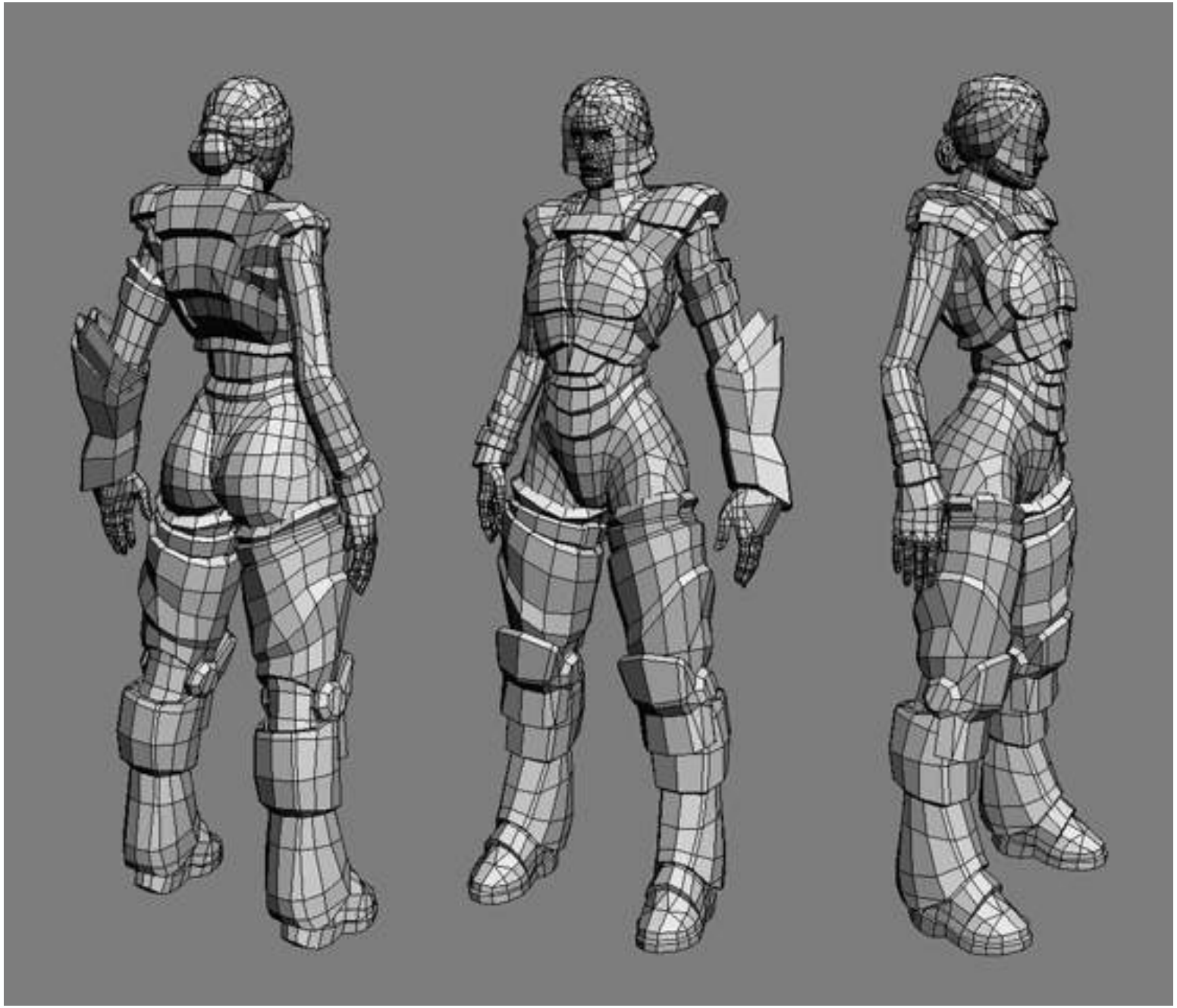
[source unknown]

© 2020 Fabio Pellacini •



[Ken Solaar]

© 2020 Fabio Pellacini •

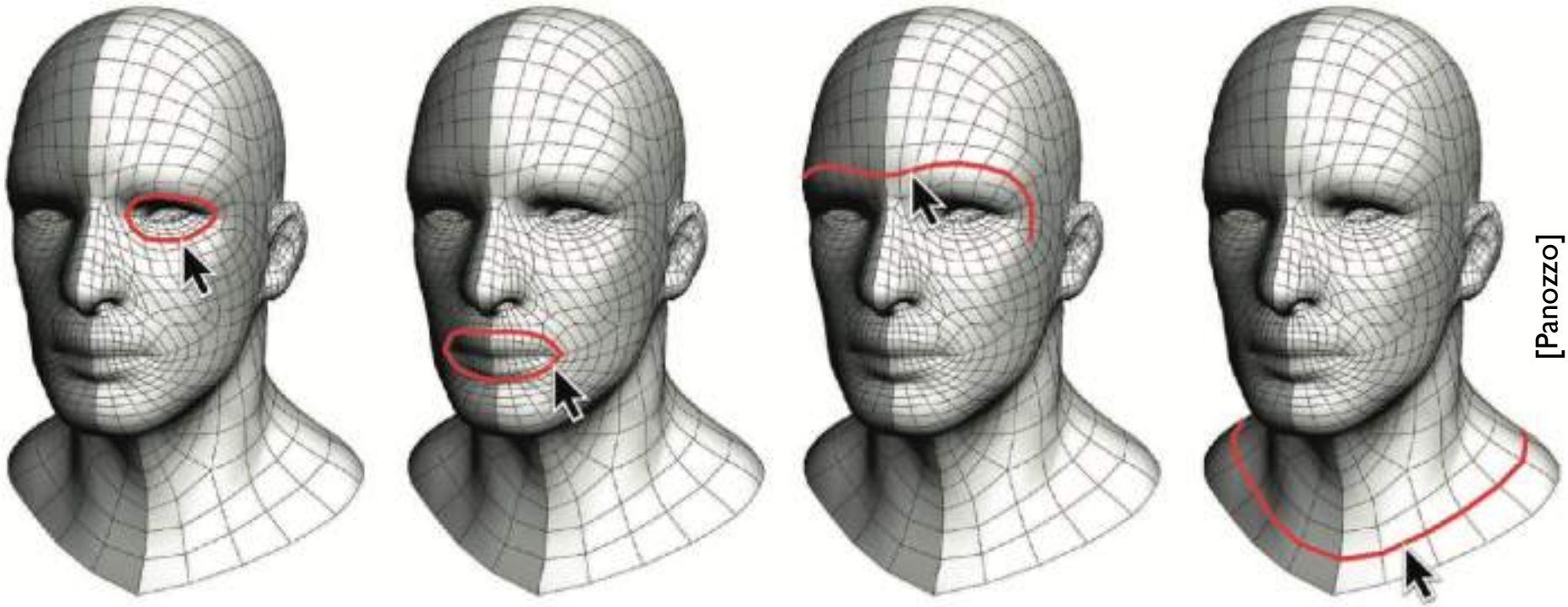


[source unknown]

© 2020 Fabio Pellacini

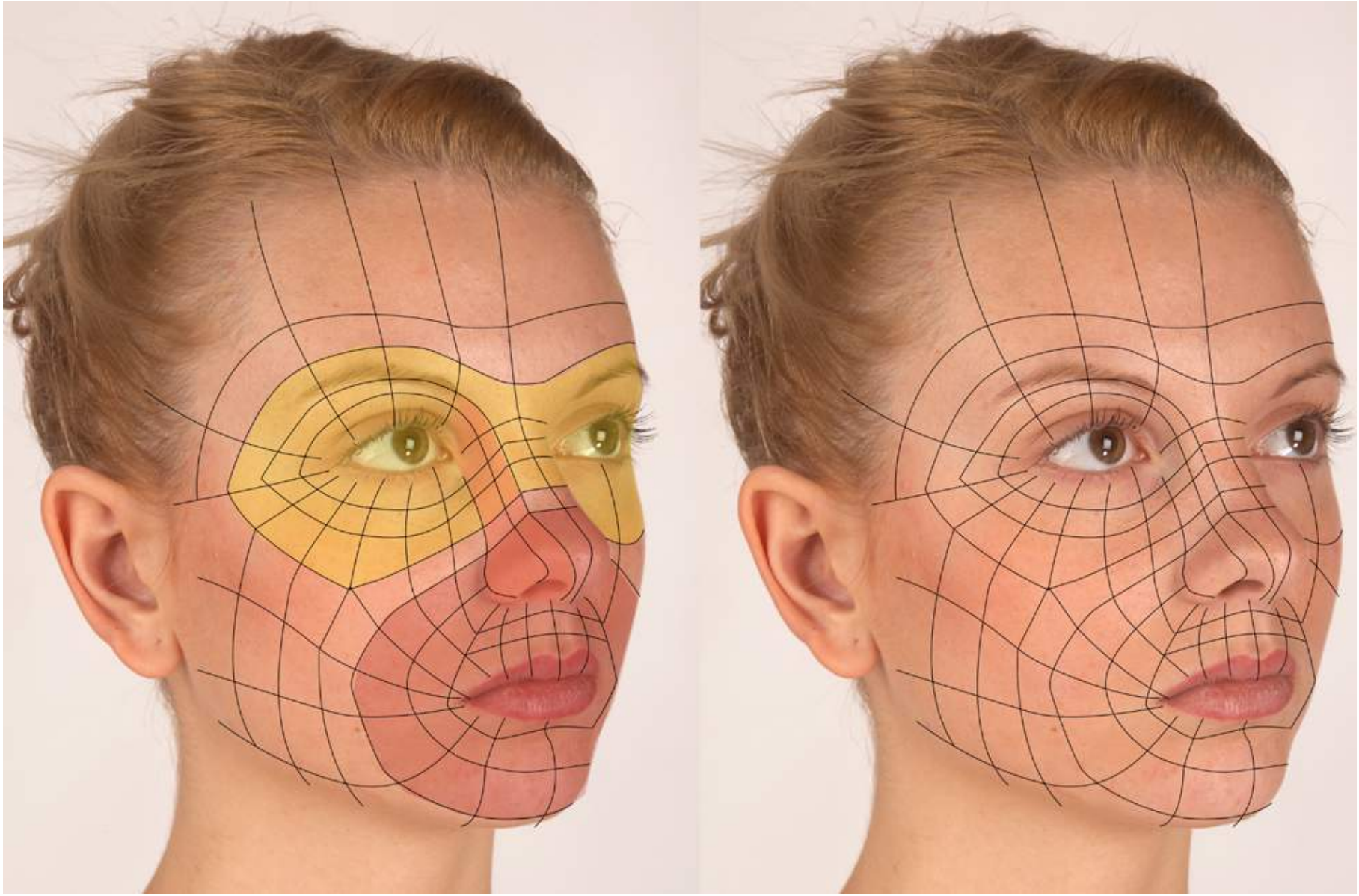
Quad meshes

- Most used modeling primitive
  - better selection and edit of edge loops
  - since it captures better "regularities" in surfaces



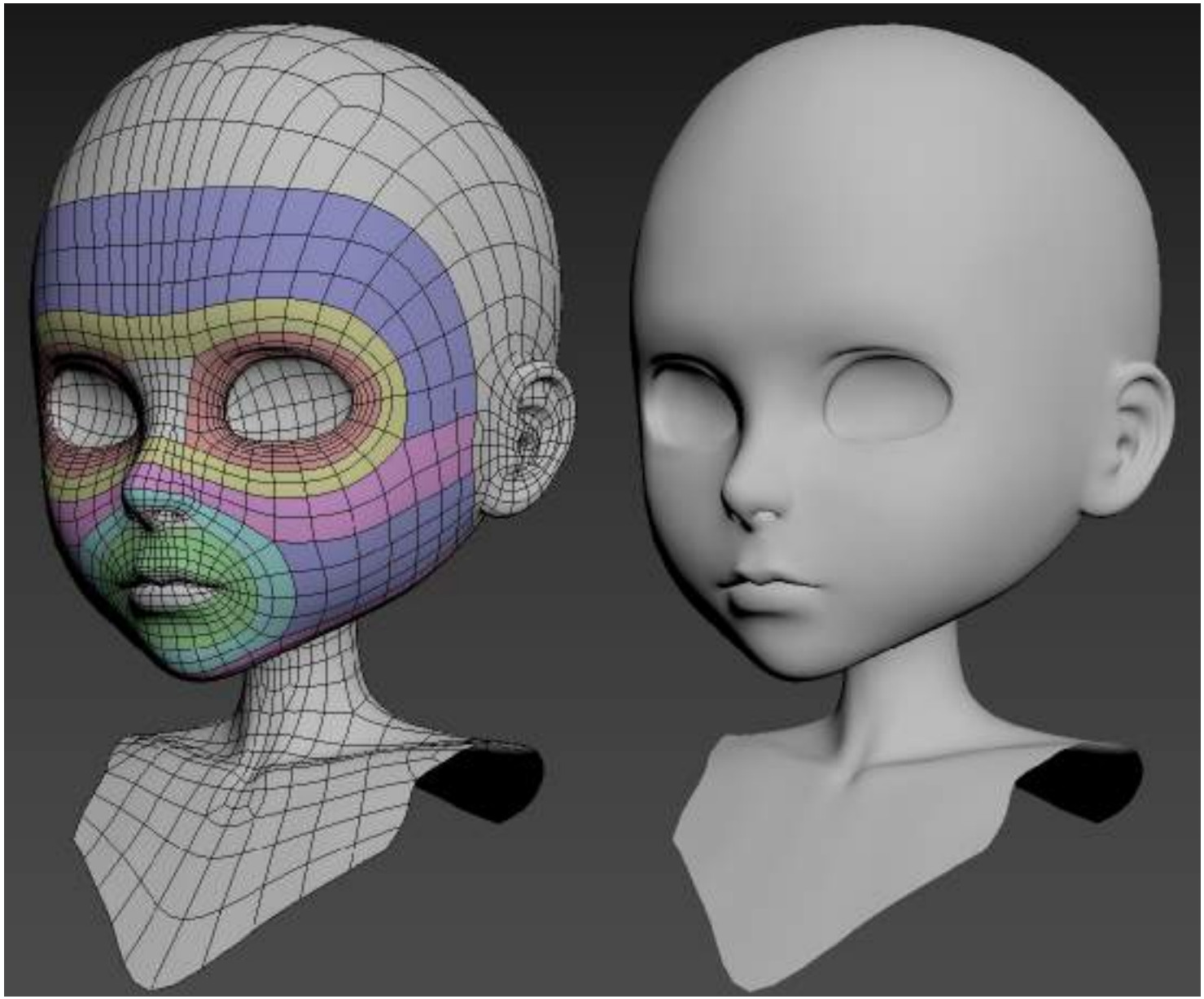
[Panozzo]

© 2020 Fabio Pellacini •



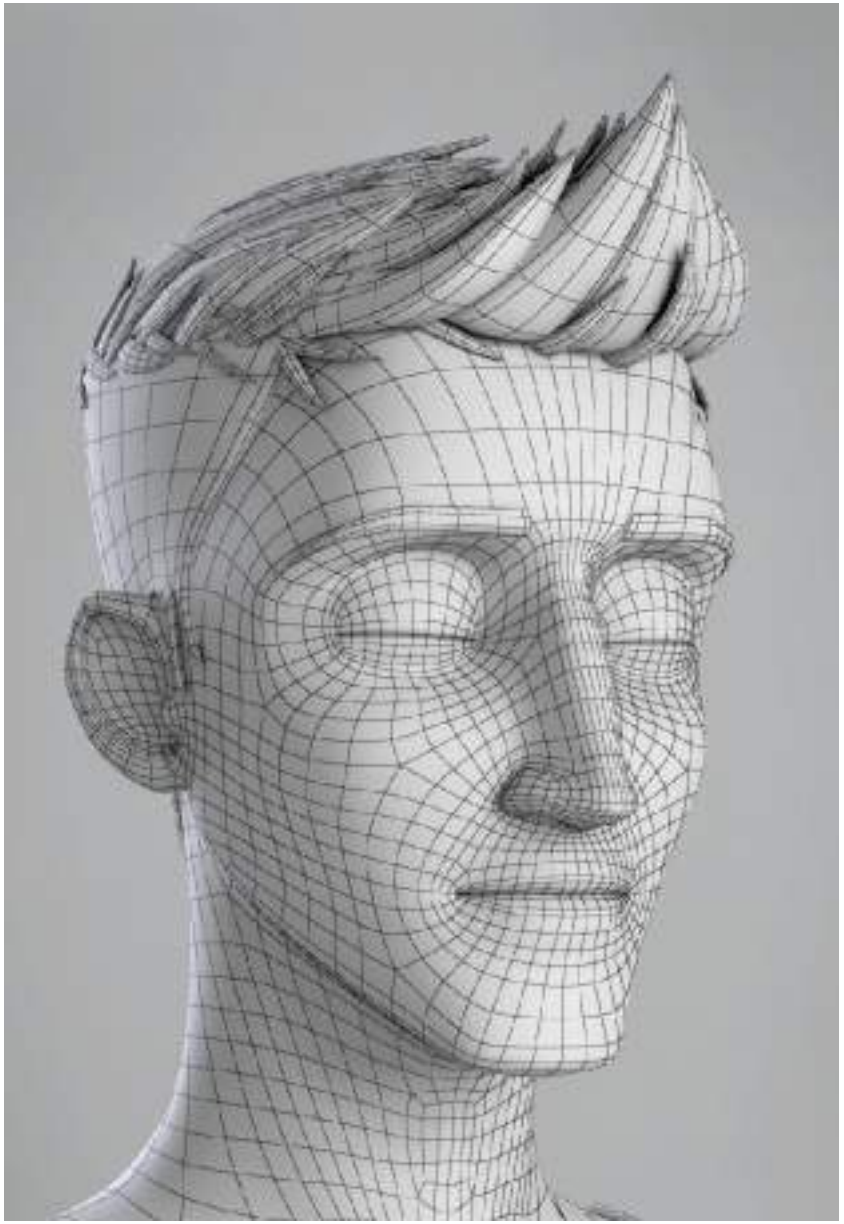
[Athey@deviantart]

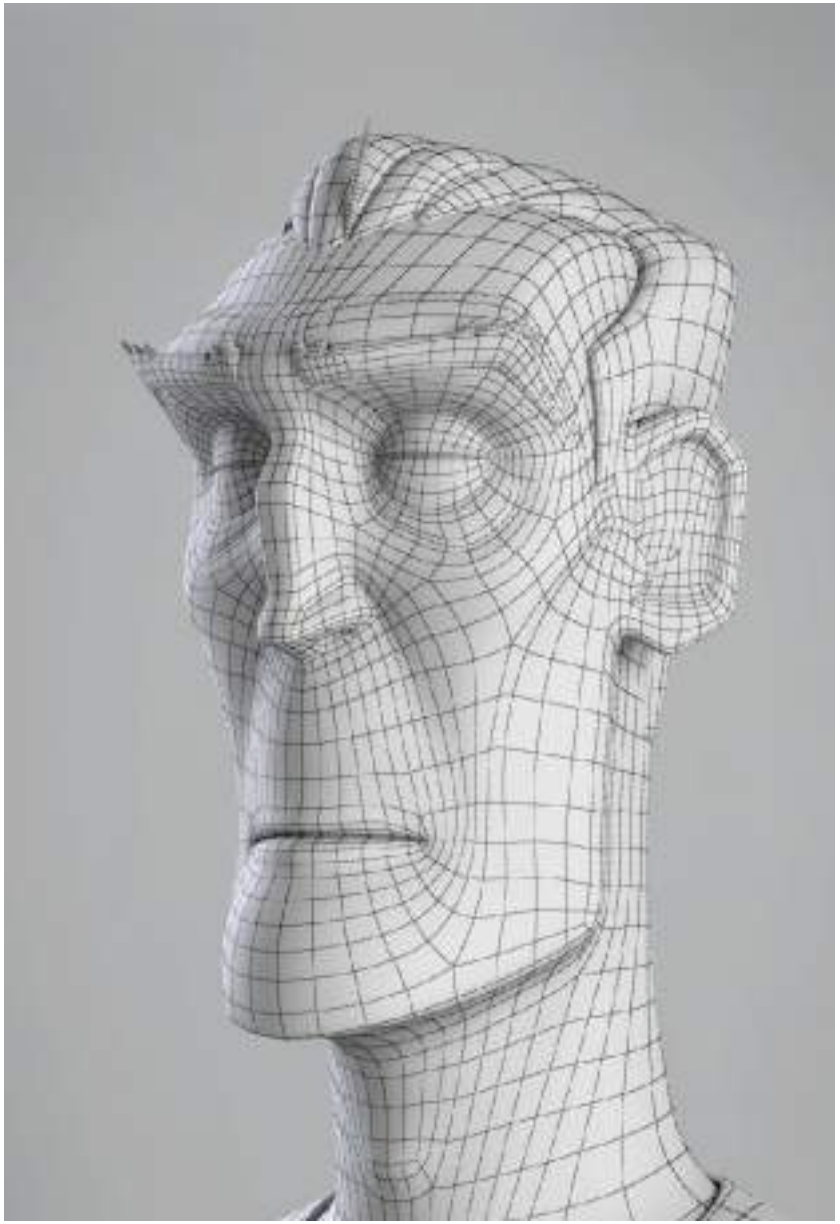
© 2020 Fabio Pellacini



[source unknown]

© 2020 Fabio Pellacini •





[Ander Liza]

© 2020 Fabio Pellacini

Parametric Surfaces

© 2020 Fabio Pellacini

Parametric surfaces

- Polygonal meshes are piecewise-flat approximations of real surfaces
- For many applications, we want to use alternative representations that are inherently smooth
- Parametric surfaces are one such representation obtained by describing a surface as a functions of 2 parameters

$$\mathbf{p}(u, v) = [x(u, v), y(u, v), z(u, v)]$$

- For example a sphere can be written parametrically as

$$\mathbf{p}(u, v) = [\cos(2\pi u)\sin(\pi v), \sin(2\pi u)\sin(\pi v), \sin(\pi v)]$$

© 2020 Fabio Pellacini

Parametric normals

- Many common surface quantities can be derived from the surface representation directly
- For example the normal at a point (u,v) is the direction orthogonal to the tangent plane, that is the plane that passed through the point with tangents as partial derivatives

$$\mathbf{n}(u,v) = \frac{\mathbf{t}_u(u,v) \times \mathbf{t}_v(u,v)}{|\mathbf{t}_u(u,v) \times \mathbf{t}_v(u,v)|}$$

$$\mathbf{t}_u(u,v) = \frac{\partial \mathbf{p}(u,v)}{\partial u}$$

$$\mathbf{t}_v(u,v) = \frac{\partial \mathbf{p}(u,v)}{\partial v}$$

© 2020 Fabio Pellacini

Surfaces patches

- As for parametric curves, many different parametric surfaces exists
- The most commonly used can be considered extensions of parametric curves called tensor product surfaces
- For example, for a Bezier curve written as

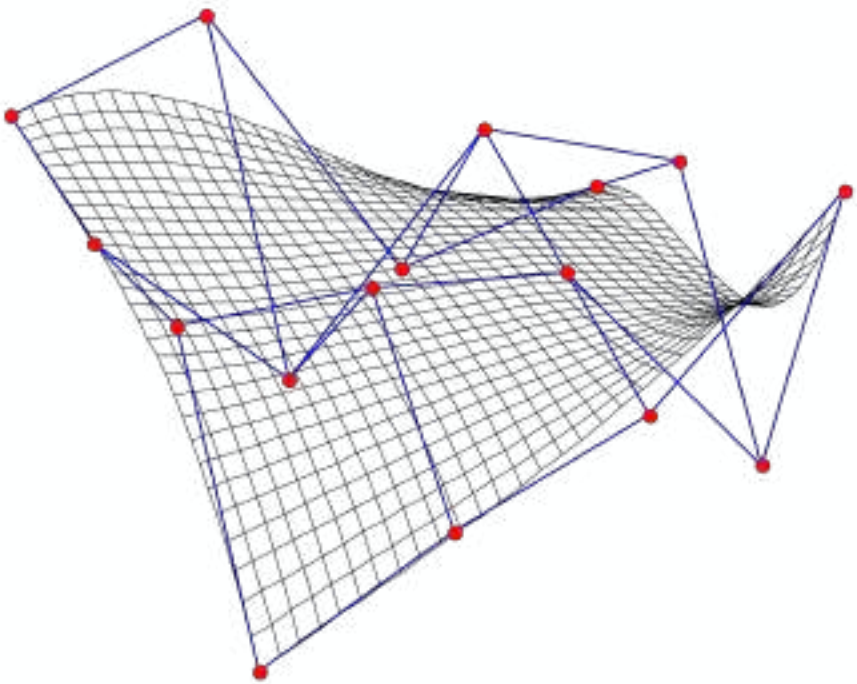
$$\mathbf{p}(u) = \sum_i b_i(u)\mathbf{p}_i$$

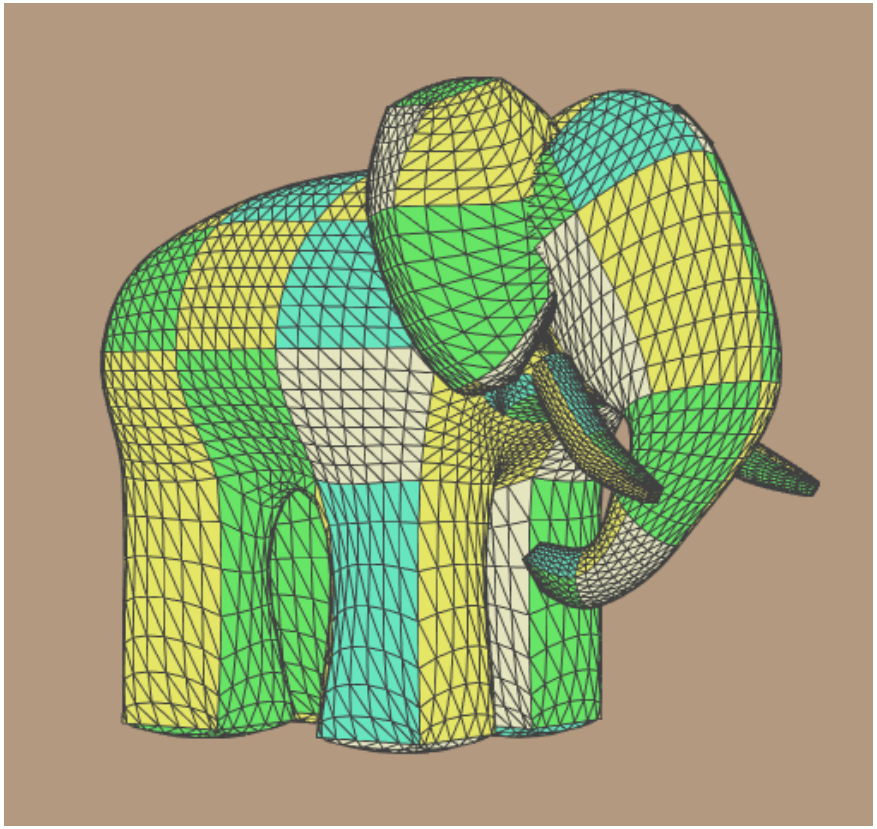
- we can derive a Bezier patch written as

$$\mathbf{p}(u, v) = \sum_i \sum_j b_i(u) b_j(v) \mathbf{p}_{i,j}$$

© 2020 Fabio Pellacini

Surface patches





[Wikipedia]

© 2020 Fabio Pellacini

NURBS

- CAD software often uses non-uniform rational B-splines, aka NURBS
- NURBS have the main advantage of being able to represent exactly conic sections, including spheres, cylinders, etc
- Other splines can only approximate these surfaces
- NURBS are rarely used outside of CAD software mostly due to the inherent complexity of their representation

© 2020 Fabio Pellacini

Subdivision Surfaces

© 2020 Fabio Pellacini

Subdivision surfaces

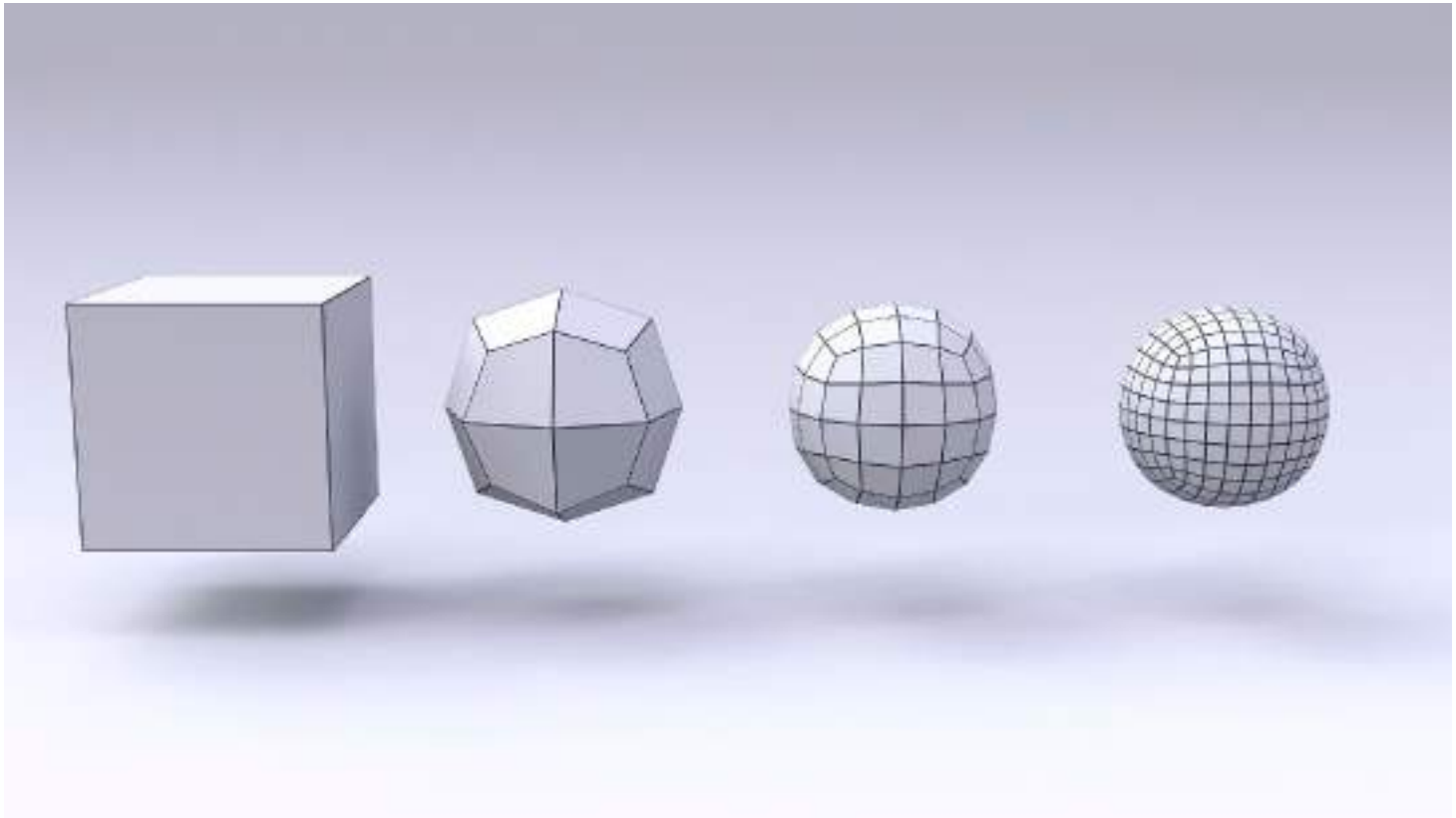
- Represent a smooth surface controlled by a polygon mesh
- Most used method for movies and soon games – works for organic and man-made models

[Panozzo]

© 2020 Fabio Pellacini

Subdivision surfaces

- Defined by recursively apply a refinement rule that add vertices and faces to the mesh and smooths out the vertex positions

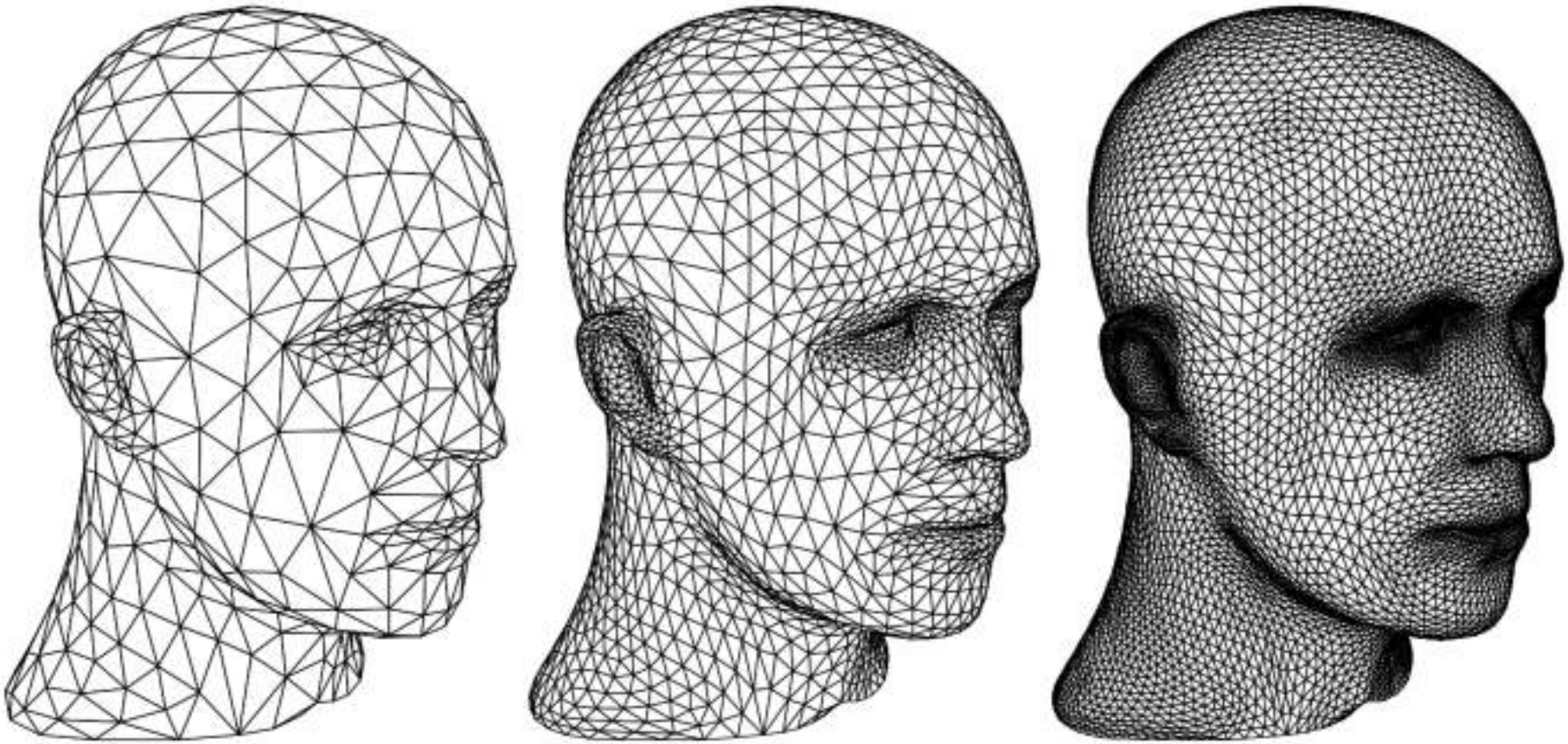


[Yining Karl]

© 2020 Fabio Pellacini

Subdivision surfaces

- Defined by recursively apply a refinement rule that add vertices and faces to the mesh and smooths out the vertex positions



[Schröder & Zorin]

© 2020 Fabio Pellacini

Subdivision surfaces – informal

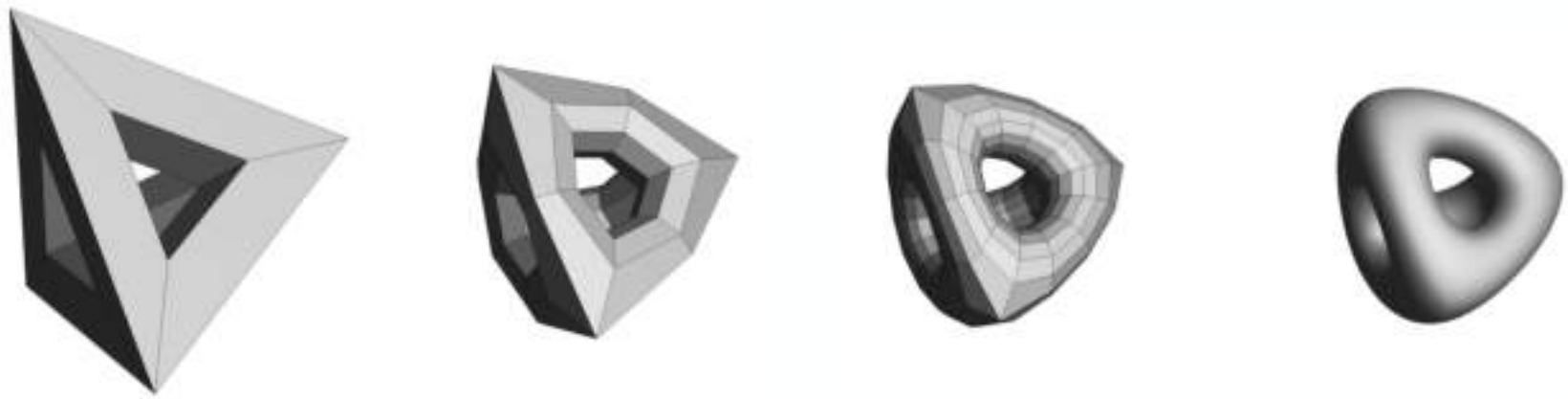
[CGGorilla/YouTube]

© 2020 Fabio Pellacini

Subdivision surfaces

- The subdivision surface is the limit surface *S* obtained by applying the subdivision rule *R* an infinite number of times to a mesh M_0

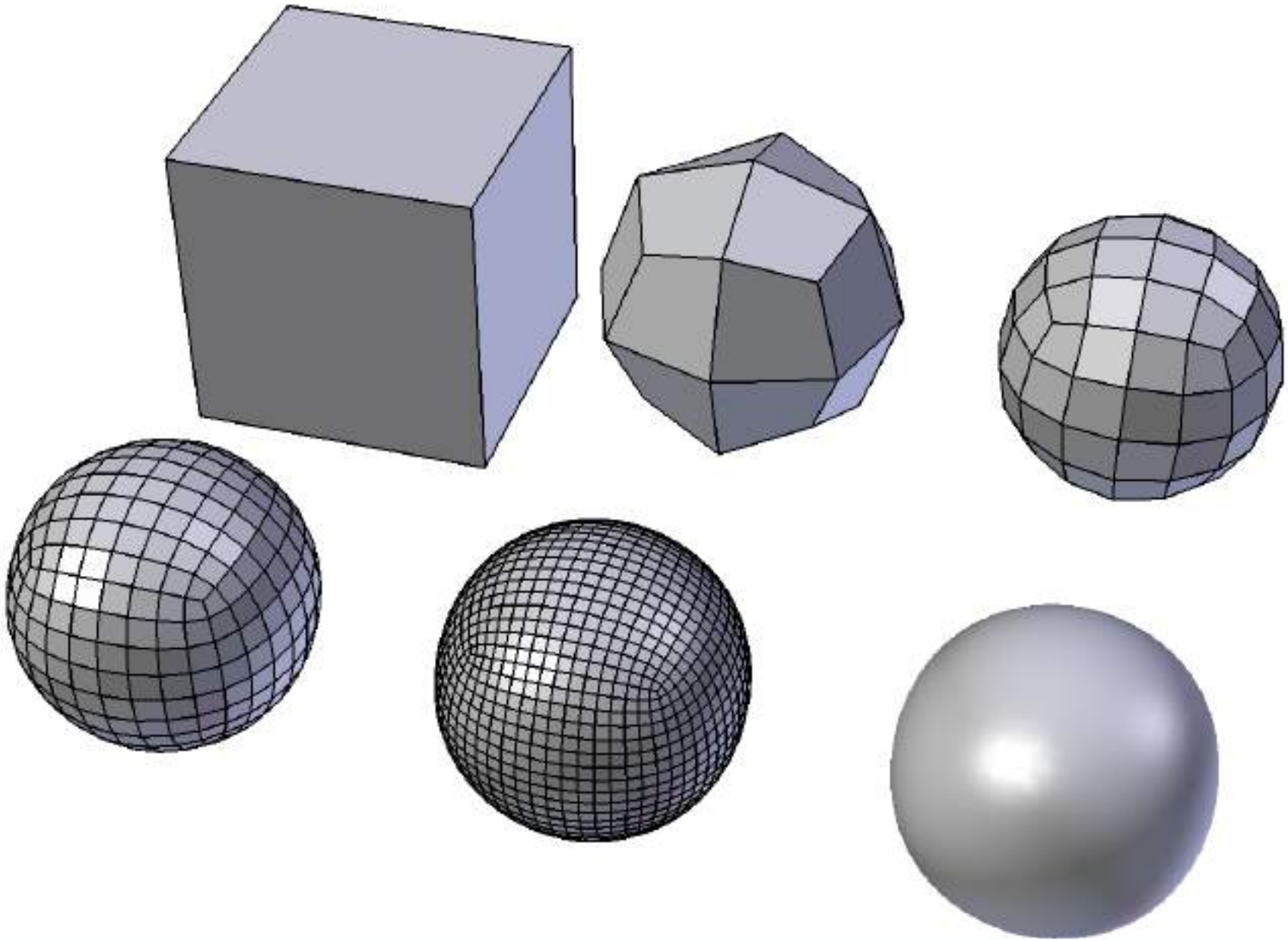
$$M_{i+1} = R(M_i) \quad S = \lim_{n\to\infty} M_n = \lim_{n\to\infty} R^n(M_0)$$



M_0 M_1 M_2 $S = M_\infty$

[de Rose et al.]

© 2020 Fabio Pellacini



© 2020 Fabio Pellacini •

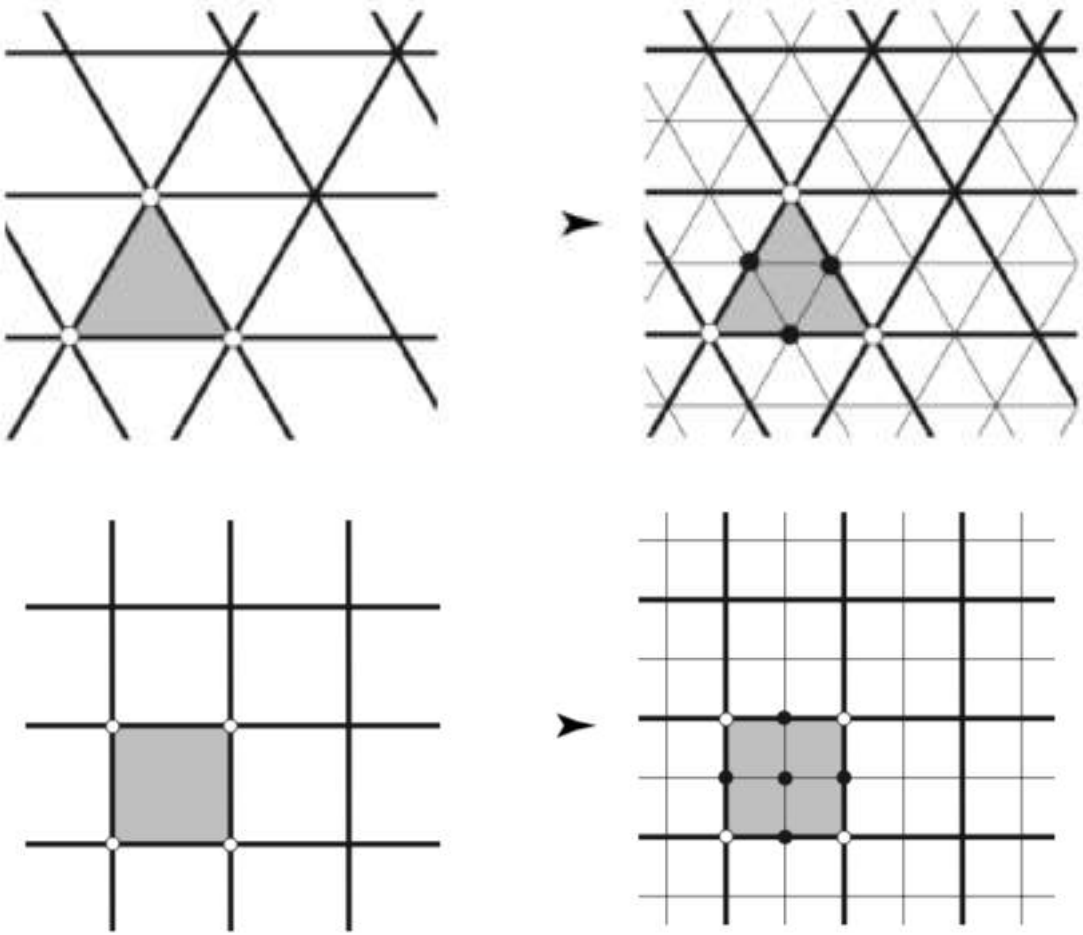
Subdivision rules

- Different rules support different input (triangles or quads) and generate surfaces with different properties
- All rules are composed of two parts
- Step 1: refine the mesh by adding vertices and faces
  - this changes the topology of the mesh
- Step 2: move refined vertices to achieve a smoother surface
  - this changes the geometry of the mesh
- We will only cover the two most common cases
  - Catmull–Clark for quad meshes
  - Loop for triangle meshes

© 2020 Fabio Pellacini

Subdivision refinement

- Edge split methods for triangle (Loop) and quads (Catmull-Clark)

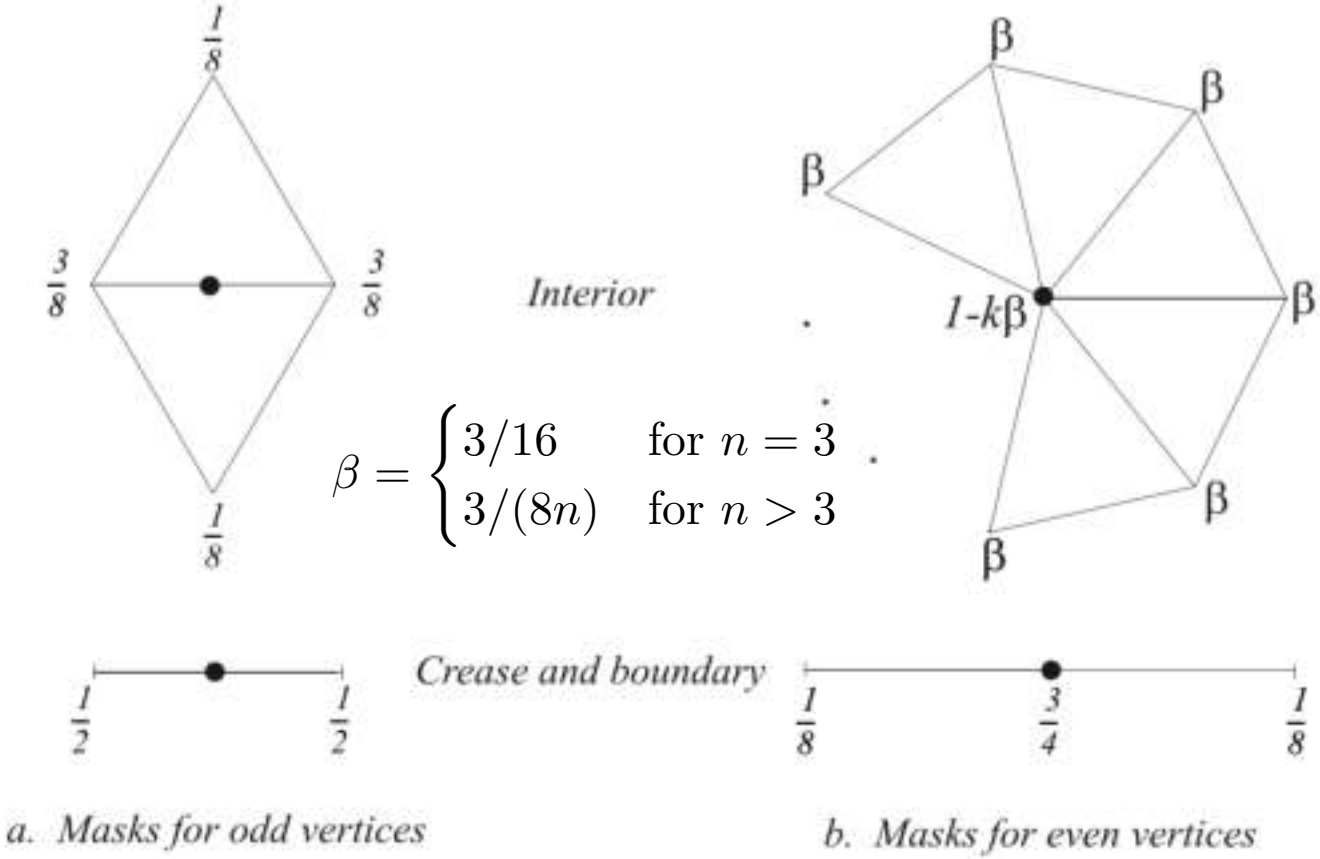


[Schröder & Zorin]

© 2020 Fabio Pellacini

Subdivision smoothing

- Loop: rules for existing vertices and vertices on the split edges
  - differentiate between interior and boundaries



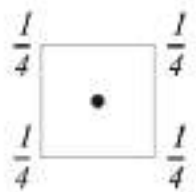
$$\beta = \begin{cases} 3/16 & \text{for } n = 3 \\ 3/(8n) & \text{for } n > 3 \end{cases}$$

[Schröder & Zorin]

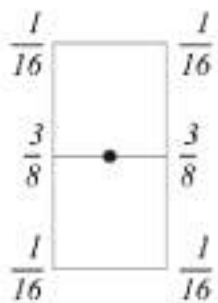
© 2020 Fabio Pellacini

Subdivision smoothing

- Catmull-Clark: rules existing and new vertices

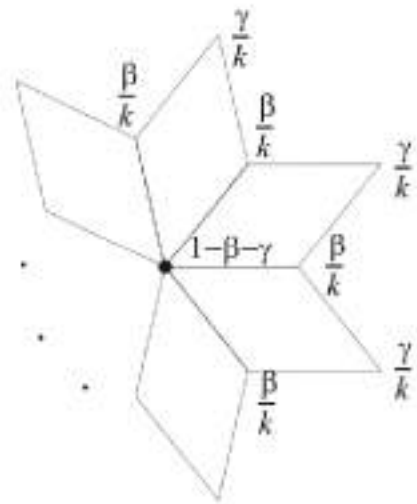


Mask for a face vertex



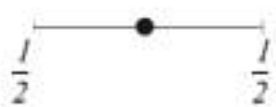
Mask for an edge vertex

Interior



$$\beta = 3/(2k) \quad \gamma = 1/(4k)$$

[Schröder & Zorin]

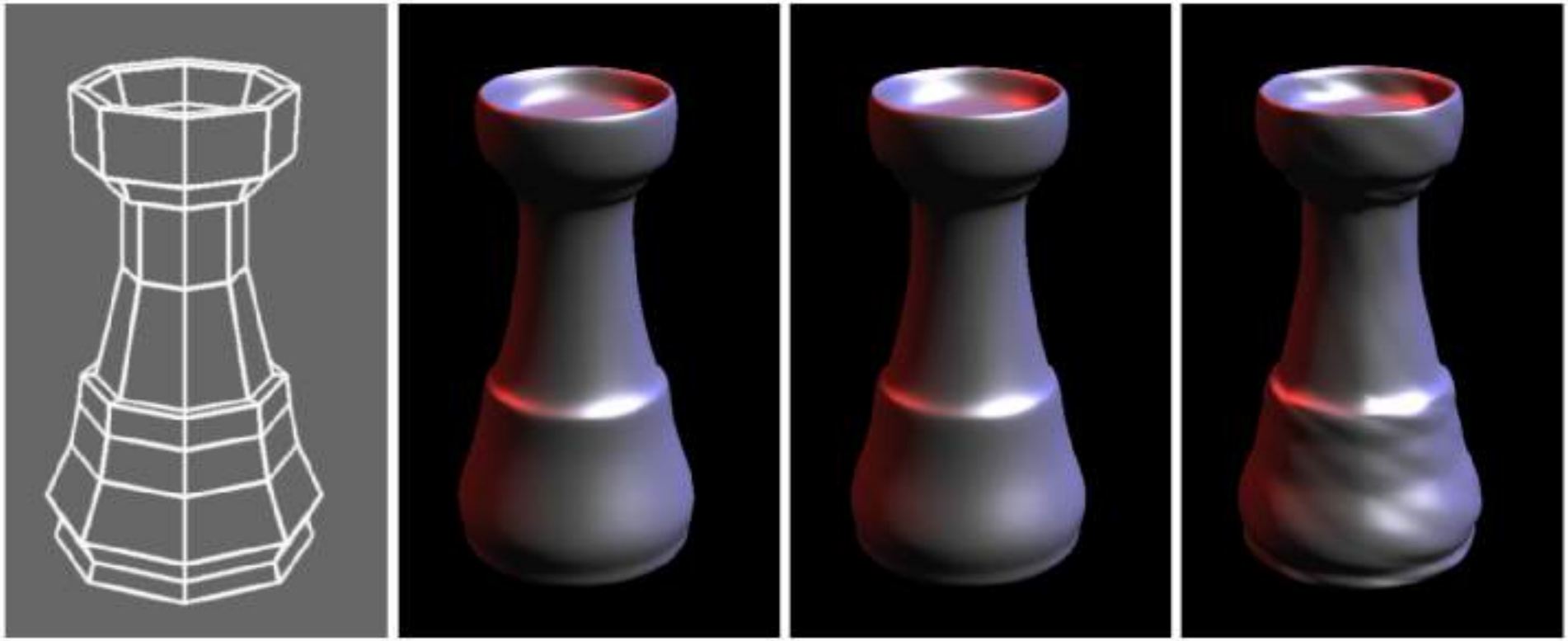


Mask for a boundary odd vertex

Crease and boundary

© 2020 Fabio Pellacini

Model comparison



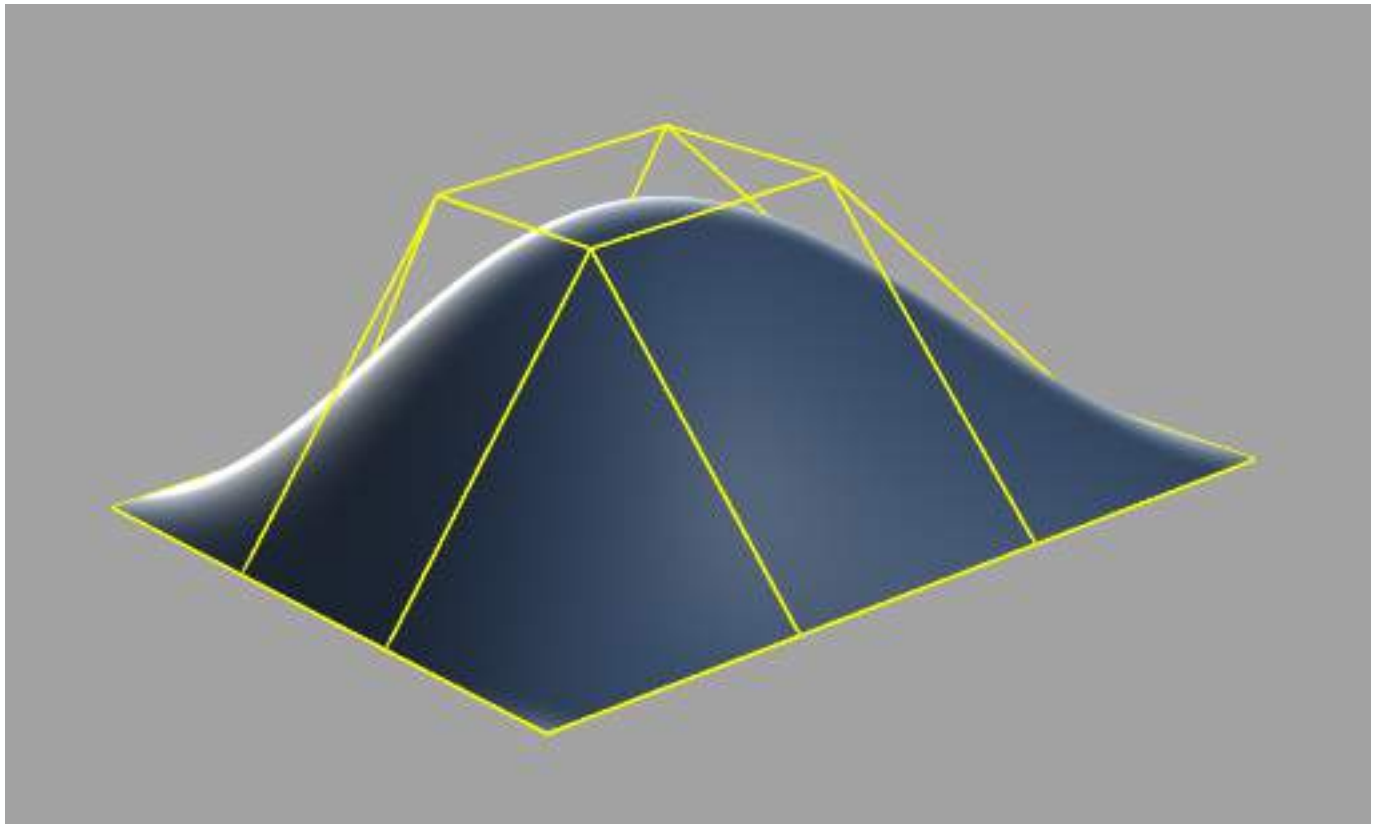
Initial mesh *Loop* *Catmull-Clark* *Catmull-Clark, after triangulation*

[Schröder & Zorin]

© 2020 Fabio Pellacini •

Creases

- Control sharpness of edges and vertices

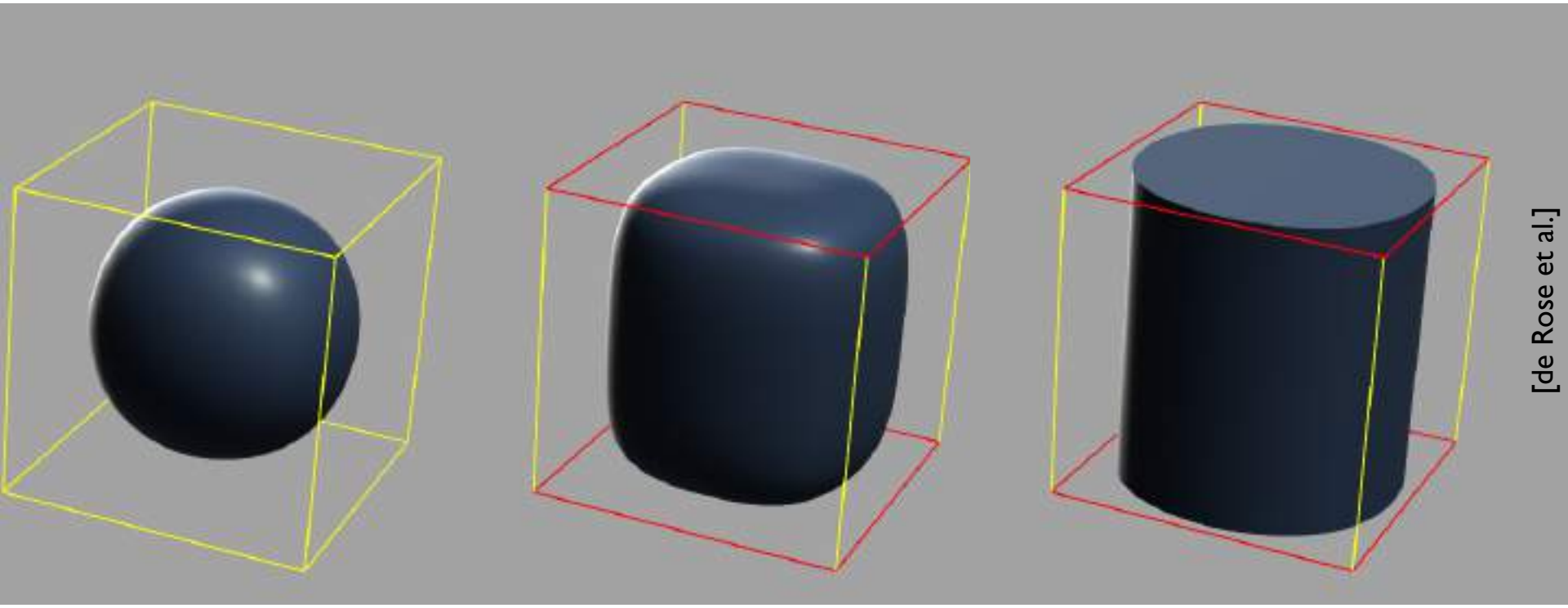


[de Rose et al.]

© 2020 Fabio Pellacini

Creases

- Control sharpness of edges and vertices

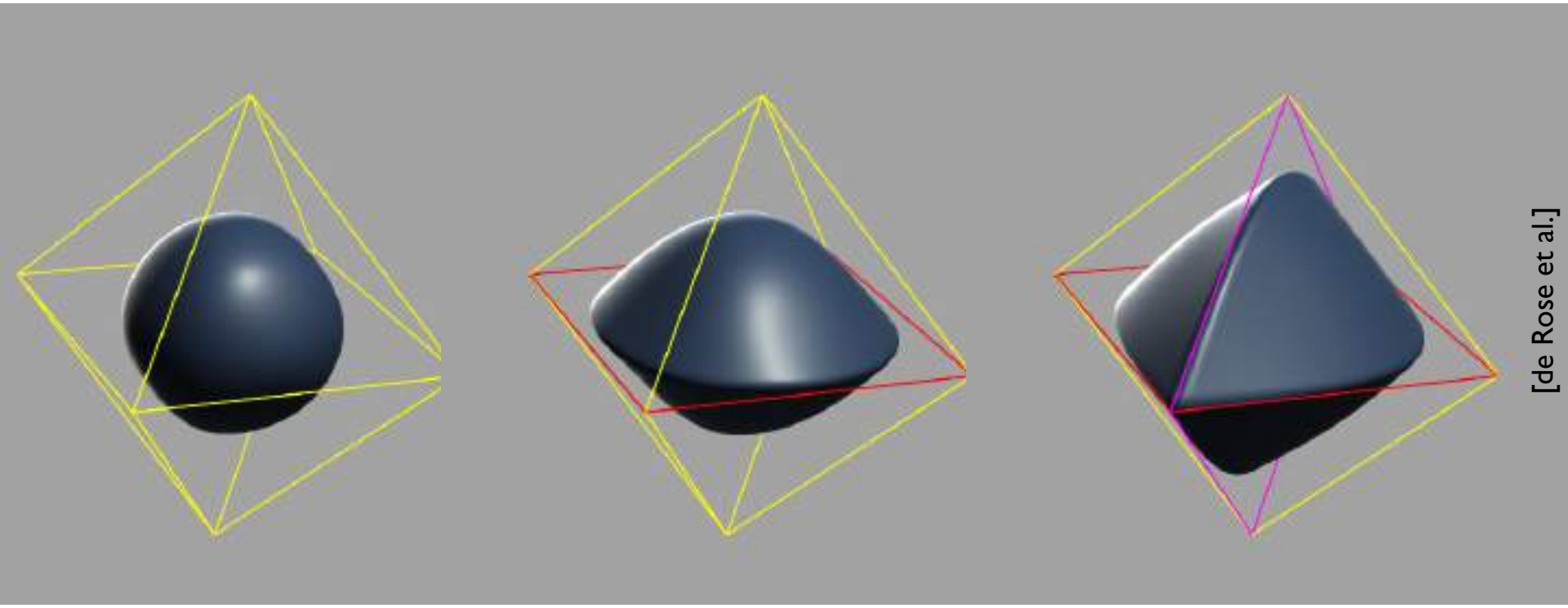


[de Rose et al.]

© 2020 Fabio Pellacini

Creases

- Control sharpness of edges and vertices

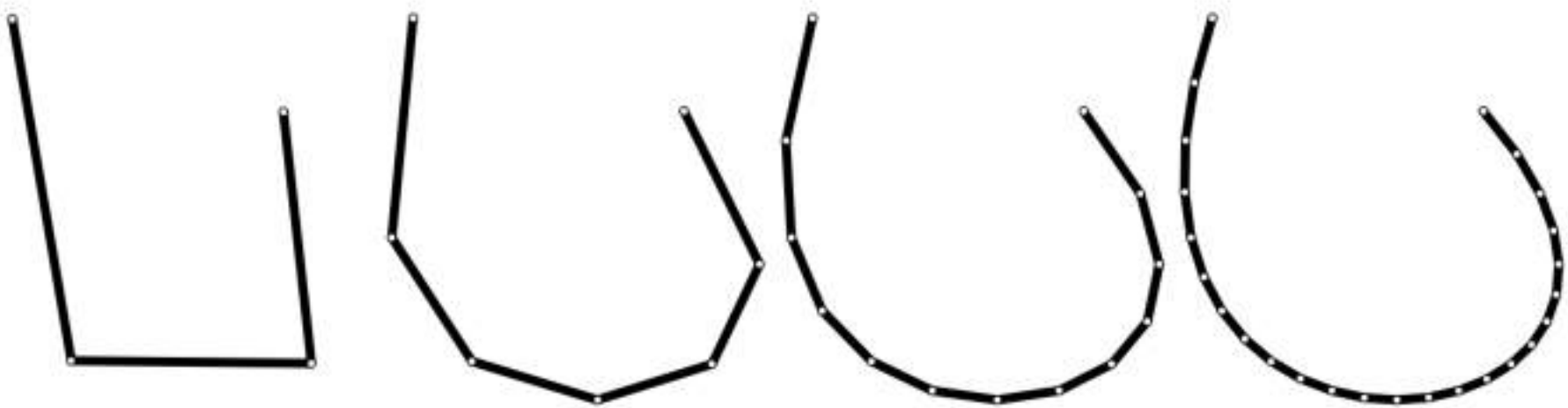


[de Rose et al.]

© 2020 Fabio Pellacini

Subdivision curves

- Main idea also works for curves
  - but less in practice since polynomials are better
- Rule: recursively split each segment in half and move vertex
- Define curves from subdivision rules, instead of polynomials
  - for B-splines we have both definitions and they are equivalent

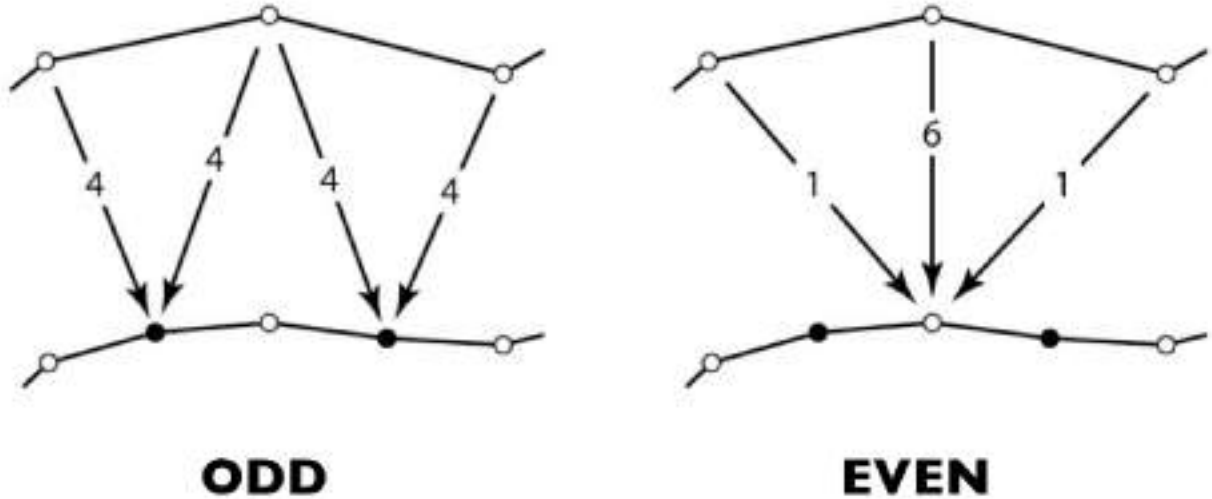


[Schröder & Zorin]

© 2020 Fabio Pellacini

Subdivision rule for B-splines

- New vertices are linear combination of old vertices



© 2020 Fabio Pellacini

Subdivision scalability

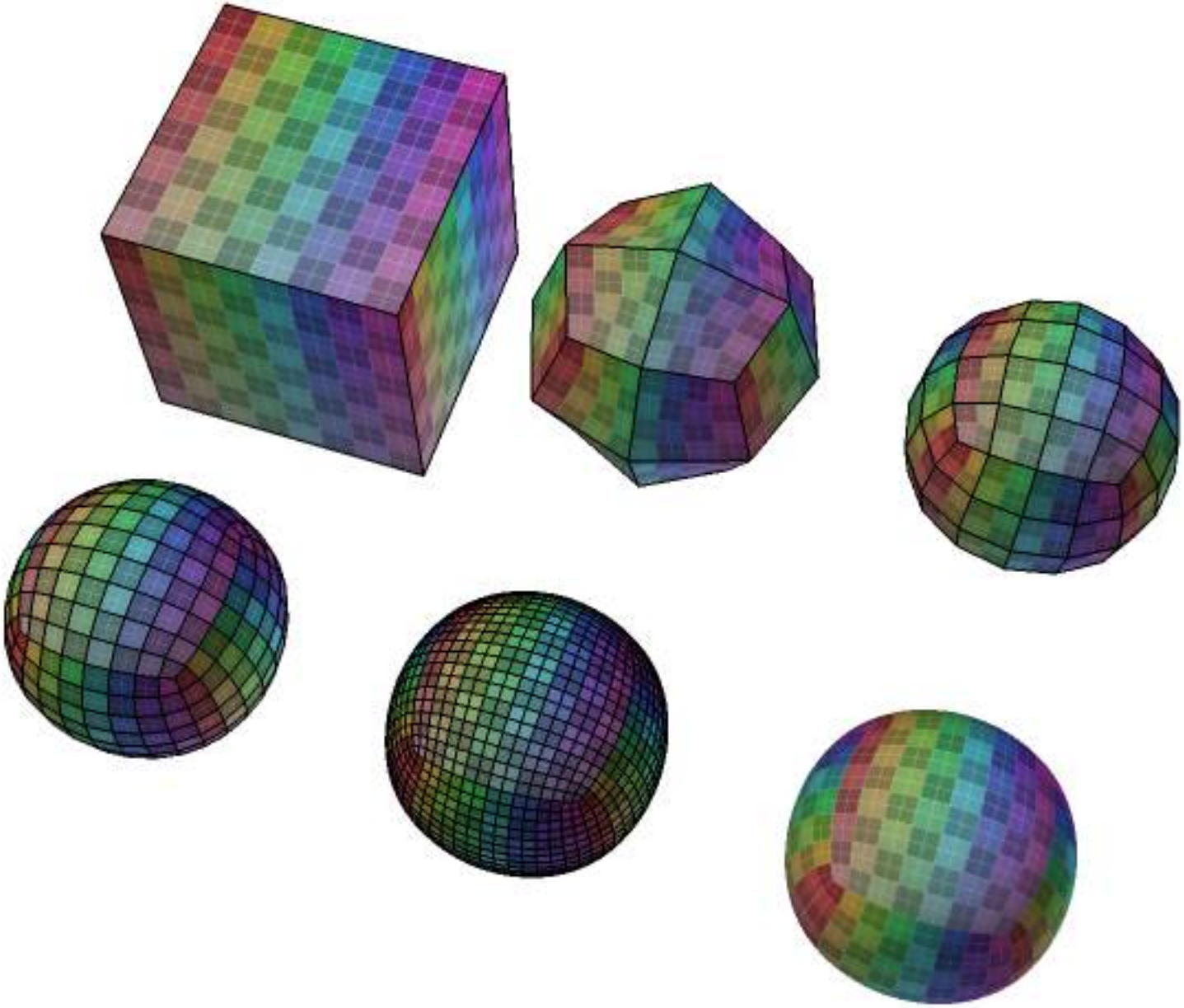
- Subdivision surfaces have many advantages when drawing shapes
- First, they can be subdivided only to the necessary level
  - further away objects are subdivided less than closer ones
  - this gives us "level of details" when rendering
- Second, they can be subdivided partially, only adding details in the regions of high curvature
  - at the expense of complex subdivision algorithms
- Third, they can be supported natively in a ray tracer with specialized intersection code that is optimized compared to simply tesselate to a fixed number of triangles

© 2020 Fabio Pellacini

Face-varying data

- So far, all data we associated with a triangle meshes was defined at the vertices
- There are cases though when associating data to faces is required, for example when representing texture coordinates
- In this case, we use the same weights we would for position, but with a different topology that has duplicated vertices when needed
- This elegant formulation allowed subdues to be used in the entertainment history

© 2020 Fabio Pellacini



© 2020 Fabio Pellacini •

Subdivision artifacts – informal

[CGGorilla/YouTube]

© 2020 Fabio Pellacini

Implementing Subdivs

© 2020 Fabio Pellacini

Implementing subdivs

- Classic implementation requires specialized data structures
  - seldomly implemented
  - often results do not match between implementations
  - considerable practical problems for exchanging data
  - special handling for creases
- Classic implementation can be done optimal though
  - view driven level of details
  - non-binary subdivision
  - accelerated raytracing without tesselating to triangles
  - check out Pixar's OpenSubdiv and Intel's Embree

© 2020 Fabio Pellacini

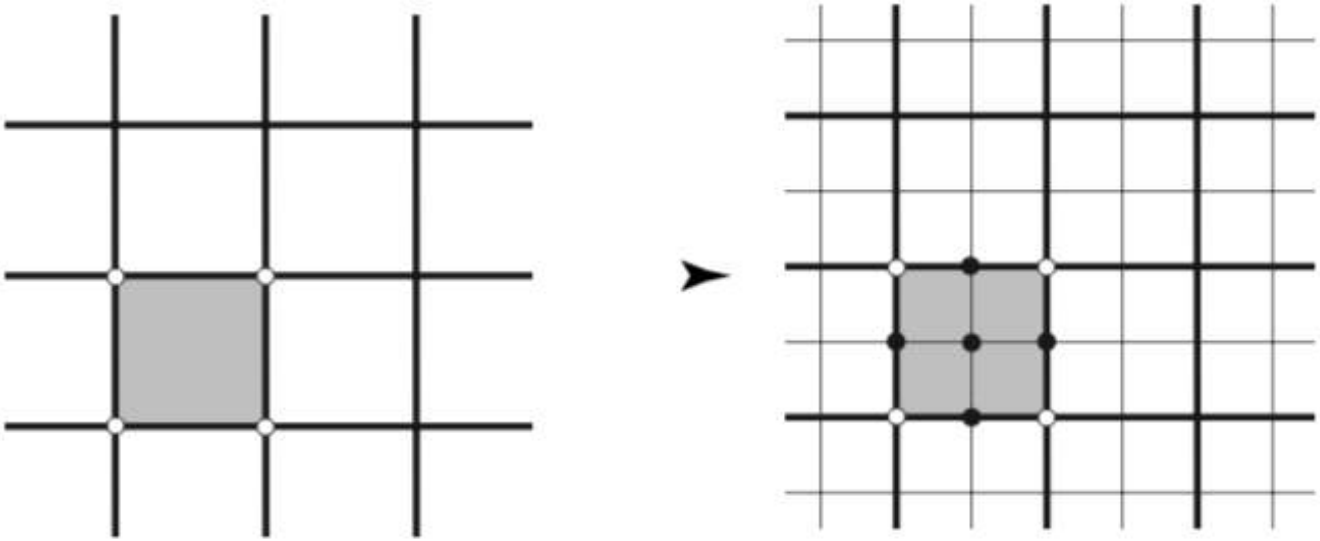
Implementing subdivs

- Simple multipass algorithm for Catmull-Clark, Loop and mixed
  - based on *A factored approach to Subdivision Surfaces* by Warren and Schaefer
  - allow for quick experimentation
- Main advatange: uses only a hash table and no specialized d.s.
- Support triangles, quads and mixed input including creases
  - with no special case handling
- Less efficient but remarkably simpler
- Will present only the Catmull-Clark variant

© 2020 Fabio Pellacini

Implementing subdivs

- Step 1: linear subivision
  - main concern: ensure that edge vertices are shared after split



[Schröder & Zorin]

© 2020 Fabio Pellacini

Implementing subdivs

- Step 1: linear subdivision
  - create one vertex for each vertex, edge and quad in original
  - vertex from vertex: set same position

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

  - vertex from edge: set in the middle between endpoints

$$\mathbf{p}'_e = \frac{\mathbf{p}_{e_1} + \mathbf{p}_{e_2}}{2}$$

  - vertex from face: set to face centroid

$$\mathbf{p}'_f = \frac{\mathbf{p}_{f_1} + \mathbf{p}_{f_2} + \mathbf{p}_{f_3} + \mathbf{p}_{f_4}}{4}$$

© 2020 Fabio Pellacini

Implementing subdivs

- Step 1: linear subdivision
  - edge vertex: two faces share an edge, but cannot duplicate the edge vertex
  - use a hash table to check whether the edge was already split
  - hash keys: pair of unordered vertex indices (edge)
    - either insert both orders or alway insert (min, max) index
  - hash values: index to the new edge

© 2020 Fabio Pellacini

Implementing subdivs

- Step 1: linear subdivision
  - create four faces *F* for each quad *f* in the original
  - use the hash table for the edge indices

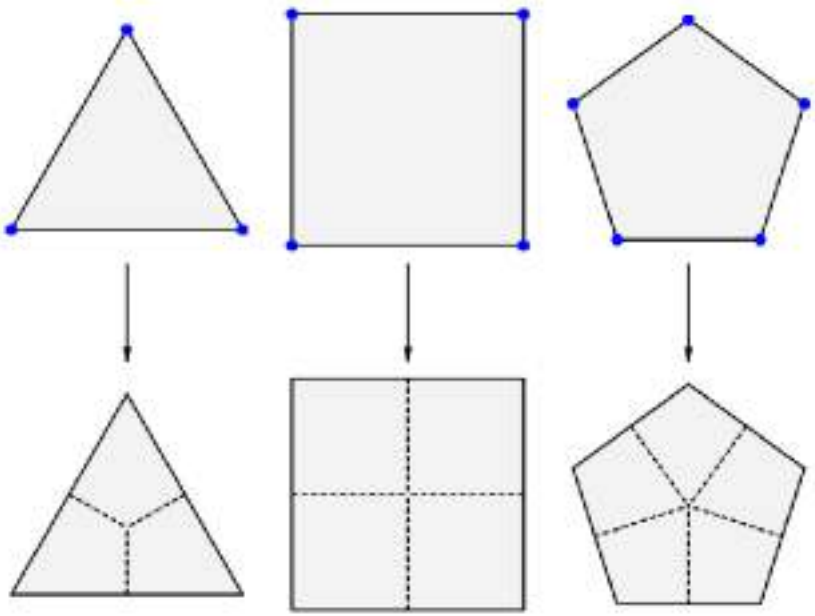
$$
\begin{aligned}
F'_{\{f,0\}} &= \{\mathbf{p}'_{f_0}, \mathbf{p}'_{e_{f_0 \to f_1}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_3 \to f_0}}\} \\
F'_{\{f,1\}} &= \{\mathbf{p}'_{f_1}, \mathbf{p}'_{e_{f_1 \to f_2}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_0 \to f_1}}\} \\
F'_{\{f,2\}} &= \{\mathbf{p}'_{f_2}, \mathbf{p}'_{e_{f_2 \to f_3}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_1 \to f_2}}\} \\
F'_{\{f,3\}} &= \{\mathbf{p}'_{f_3}, \mathbf{p}'_{e_{f_3 \to f_0}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_2 \to f_3}}\}
\end{aligned}
$$

- Works also for non-quad faces

© 2020 Fabio Pellacini

Implementing subdivs

- Step 1: linear subdivision
  - works also for non-quad faces as input
  - after one pass all faces are quads

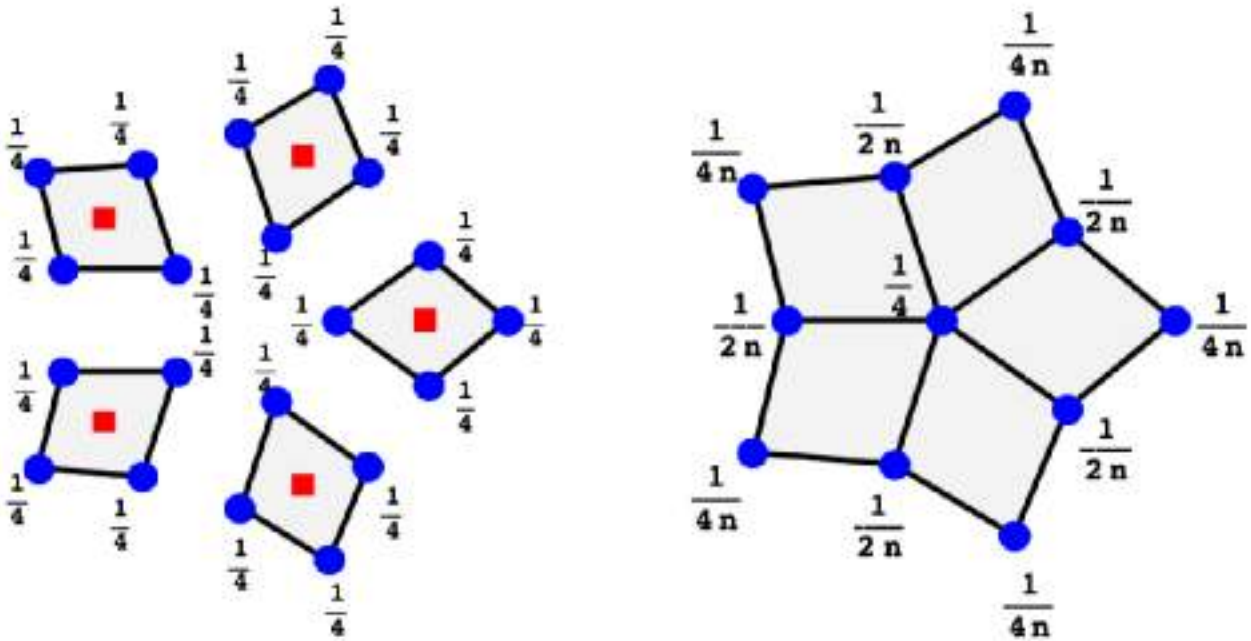


[Warren & Shaefer]

© 2020 Fabio Pellacini

Implementing subdivs

- Step 2: averaging pass
  - smooth vertices based on their neighbors
  - update vertex position as the average of the centroids of the faces touching it



[Warren & Shaefer]

© 2020 Fabio Pellacini

Implementing subdivs

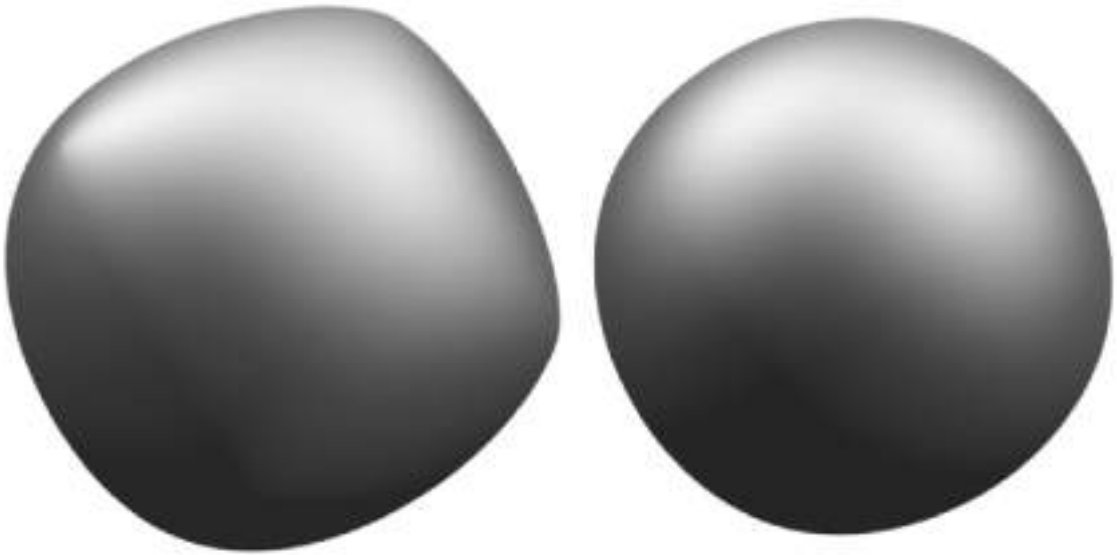
- Step 2: averaging pass
  - smooth vertices based on their neighbors
  - update vertex position as the average of the centroids of the faces touching it
  - implemented just like smoothing normals

$$\mathbf{p}_i'' = \frac{\sum_{F \in adj(\mathbf{p}_i)} \text{centroid}(F)}{\sum_{F \in adj(\mathbf{p}_i)} 1}$$

© 2020 Fabio Pellacini

Implementing subdivs

- Step 3: correction pass
  - need to correct for vertices that have non-standard number of adjacent faces



[Warren & Shaefer]

© 2020 Fabio Pellacini

Implementing subdivs

- Step 3: correction pass
  - need to correct for vertices that have non-standard number of adjacent faces
  - correct vertex position after linear subdivision with vertex position after averaging

$$\mathbf{p}_i''' = \mathbf{p}_i' + (\mathbf{p}_i'' - \mathbf{p}_i') \frac{4}{\sum_{F \in adj(\mathbf{p}_i)} 1}$$

© 2020 Fabio Pellacini

Implementing subdivs

- Handling boundaries
  - boundaries use different rules to fix them during subdivision
  - the previous algorithm can be easily modified to account for that
  - boundaries edges can be detected with the edge dictionary
  - we tag each subdivided vertex as either being a boundary vertex, belonging to a binary edge or otherwise
  - for boundary vertices, no averaging is performed
  - for vertices on boundary edges, we average the edge vertices
  - for all other vertices, we apply the corrections before
- Handling texture coords
  - for texture coords we lock boundary vertices to maintain discontinuities

© 2020 Fabio Pellacini

Implementing subdivs

- A possible implementation follows the three phases described before
  - Takes as input an array of quads, an array of vertex coordinates and a flag to indicate whether the boundary should move

```
pair<vector<vec4i>, vector<vec3f>>
 subdivide_catmullclark(vector<vec4i> quads,
            vector<vec3f> verts, bool lock_boundary) {
  <initialize edges>
  <create vertices> // phase 1
  <create faces>
  <setup boundary>
  <averaging>       // phase 2
  <correction>      // phase 3
  return {tquads, tverts};
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- During initialization, we create the edge dictionary

```
<initialize edges> {
  // construct edge map and get edges and boundary
  auto emap     = make_edge_map(quads);
  auto edges    = get_edges(emap);
  auto boundary = get_boundary(emap);
  // initialize number of elements
  auto nv = (int)vert.size();
  auto ne = (int)edges.size();
  auto nb = (int)boundary.size();
  auto nf = (int)quads.size();
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- We can build an edge map by storing each edge with indices sorted
- Note: the implementation here is not efficient

```
struct edge_map {
  hash_map<vec2i, int> index  = {}; // edge indices
  vector<vec2i>        edges  = {}; // list of edges
  vector<int>          nfaces = {}; // list of boundaries
};

edge_map make_edge_map(vector<vec4i> quads) {
  auto emap = edge_map{};
  for (auto q : quads) {
    insert_edge(emap, {q.x, q.y});
    insert_edge(emap, {q.y, q.z});
    if (q.z != q.w) insert_edge(emap, {q.z, q.w});
    insert_edge(emap, {q.w, q.x});
  }
  return emap;
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- We can build an edge map by storing each edge with indices sorted
- Note: the implementation here is not efficient

```
void insert_edge(edge_map& emap, vec2i edge) {
  auto key = vec2i{min(edge), max(edge)};
  if (emap.indices.contains(key)) {
    emap.nfaces[emap.index[key]] += 1;
  } else {
    auto idx = (int)emap.edges.size();
    emap.index[key] = idx;
    emap.edges.push_back(key);
    emap.nfaces.push_back(1);
 }
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- Then we first initialize the array of vertices and make a vertex for each vertex, edge and face in the original mesh

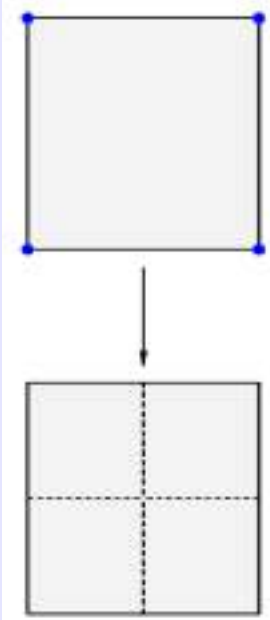
```
<create vertices> {
  auto tverts = vector<vec3f>();
  for (auto v : verts) // vertices [0, nv)
    tverts.push_back(v);
  for (auto e : edges) // edge vertices [nv, nv+ne)
    tverts.push_back((verts[e.x] + verts[e.y]) / 2);
  for (auto q : quads) // face vertices [nv+ne, nv+ne+nf)
    if (q.z != q.w)    // quads
      tverts.push_back((verts[q.x] + verts[q.y] +
                        verts[q.z] + verts[q.w]) / 4);
    else               // triangles
      tverts.push_back((verts[q.x] + verts[q.y] +
                        verts[q.z]) / 3);
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- Then we create faces by splitting each face into 3 or 4 quads since here we support both triangles and quads

```
<create faces> {
  auto tquads = vector<vec4i>();
  for (auto [i, q] : enumerate(quads)) {
    if(q.z != q.w) {
      tquads.push_back({q.x,nv+emap[q.x,q.y],
            nv+ne+i, nv+emap[q.w, q.x]});
      tquads.push_back({q.y,nv+emap[q.y,q.z],
            nv+ne+i, nv+emap[q.x, q.y]});
      tquads.push_back({q.z, nv+emap[q.z, q.w],
            nv+ne+i, nv+emap[q.y, q.z]});
      tquads.push_back({q.w, nv+emap[q.w, q.x],
            nv+ne+i, nv+emap[q.z, q.w]});
    } else { … }
  }
}
```

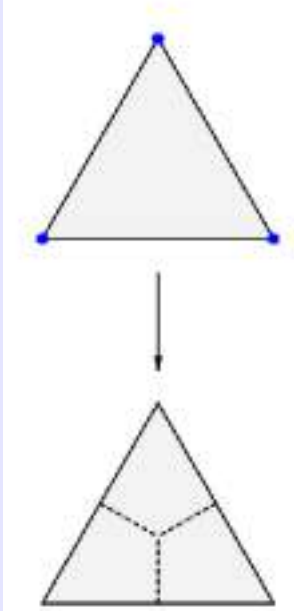


© 2020 Fabio Pellacini

Implementing subdivs

- Then we create faces by splitting each face into 3 or 4 quads since here we support both triangles and quads

```
<create faces> {
  auto tquads = vector<vec4i>();
  for (auto [i, q] : enumerate(quads)) {
    if(q.z != q.w) {
      …
    } else {
      tquads.push_back({q.x, nv+emap[q.x, q.y],
            nv+ne+i, nv+emap[q.z, q.x]});
      tquads.push_back({q.y, nv+emap[q.y, q.z],
            nv+ne+i, nv+emap[q.x, q.y]});
      tquads.push_back({q.z, nv+emap[q.z, q.x],
            nv+ne+i, nv+emap[q.y, q.z]});
    }
  }
}
```



© 2020 Fabio Pellacini

Implementing subdivs

- We then setup the boundary arrays to track the indices of boundary vertices

```
<setup boundary> {
 auto tboundary = vector<vec2i>();
  for (auto e : boundary) {
    tboundary.push_back({e.x, nv+emap[e]});
    tboundary.push_back({nv+emap[e], e.y});
  }
  auto tcrease_edges = vector<vec2i>();
  auto tcrease_verts = vector<int>();
  if (lock_boundary) {
    for (auto& b : tboundary) {
      tcrease_verts.push_back(b.x);
      tcrease_verts.push_back(b.y);
    }
  } else {
    for (auto& b : tboundary) tcrease_edges.push_back(b);
  }
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- To apply subdivision rules we need to track whether a vertex comes from a boundary vertex, a boundary edge, or otherwise
- We do so by tracking vertex valence which is 0 for crease vertices, 1 for boundary edges, or 2 otherwise

```
<setup boundary> {
  auto tverts_val = vector<int>(tverts.size(), 2);
  for (auto& e : tboundary) {
    tvert_val[e.x] = (lock_boundary) ? 0 : 1;
    tvert_val[e.y] = (lock_boundary) ? 0 : 1;
  }
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- We then apply averaging rules

```
<averaging> {
  auto avert  = vector<vec3f>(tvert.size(), zero3f);
  auto acount = vector<int>(tvert.size(), 0);
  for (auto p : tcrease_verts) {
    if (tvert_val[p] != 0) continue;
    avert[p] += tverts[p]; acount[p] += 1;
  }
  for (auto& e : tcrease_edges) {
    auto c = (tverts[e.x] + tverts[e.y]) / 2;
    for (auto vid : e) {
      if (tverts_val[vid] != 1) continue;
      avert[vid] += c; acount[vid] += 1;
    }
  }
  …
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- We then apply averaging rules

```
<averaging> {
  …
 for (auto& q : tquads) {
   auto c = (tverts[q.x] + tverts[q.y] +
             tverts[q.z] + tverts[q.w]) / 4;
   for (auto vid : q) {
     if (tverts_val[vid] != 2) continue;
     avert[vid] += c; acount[vid] += 1;
   }
 }
 for (auto i : range(tverts))
   avert[i] /= (float)acount[i];
}
```


© 2020 Fabio Pellacini

Implementing subdivs

- Finally we perform the correction pass

```
<correction> {
  for (auto i : range(tverts)) {
    if (tvert_val[i] != 2) continue;
    averts[i] = tverts[i] +
      (averts[i] - tverts[i]) * (4 / (float)acount[i]);
  }
  tverts = averts;
}
```


© 2020 Fabio Pellacini

Displacements and Bumps

© 2020 Fabio Pellacini

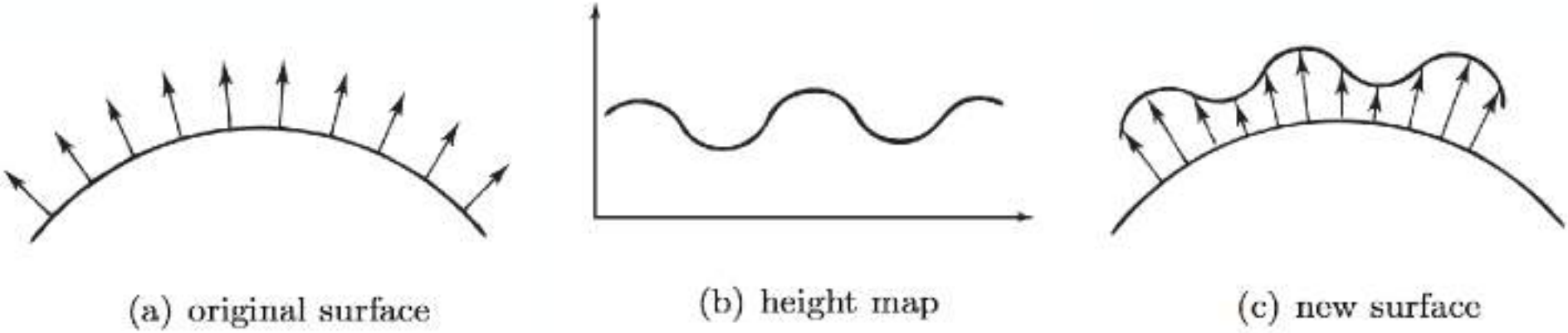
Adding details back

- Subdivision surfaces are a controllable and smooth making them the most common method for modeling smooth shapes
- But subdivs lack fine scale details that detailed meshes have
- To get details, we could increase significantly the resolution for the control mesh
- But this would be essentially make us go back to the same vertex density of triangle/quad meshes, with the added inefficiency of having a subdiv
- Instead we use images to control geometric details, just like textures control material properties
- This is also helpful when rendering triangle meshes on the GPU

© 2020 Fabio Pellacini

Displacement mapping

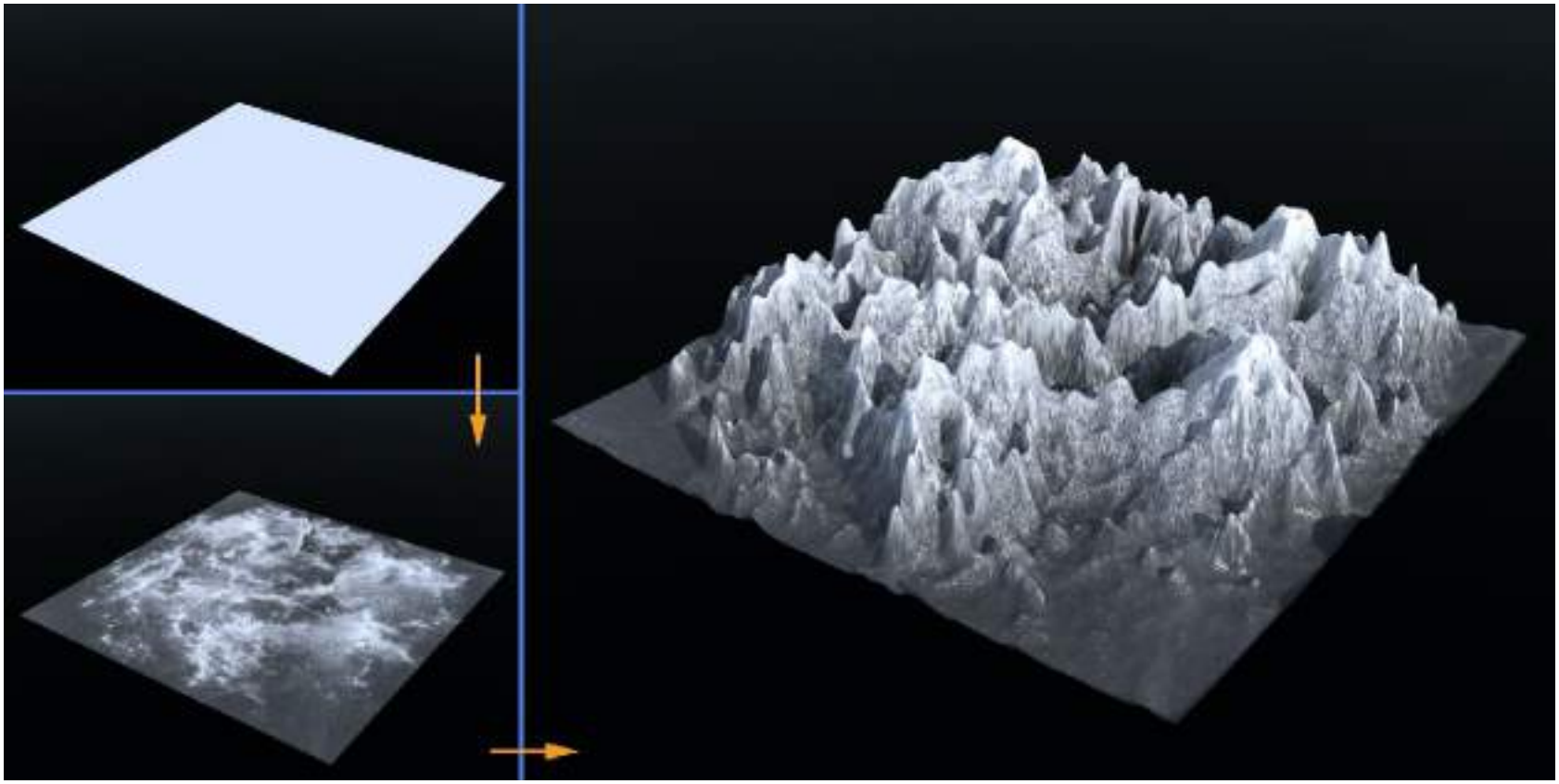
- Displacement mapping: use a texture that defines the displacement along the normal direction at each surface location



© 2020 Fabio Pellacini

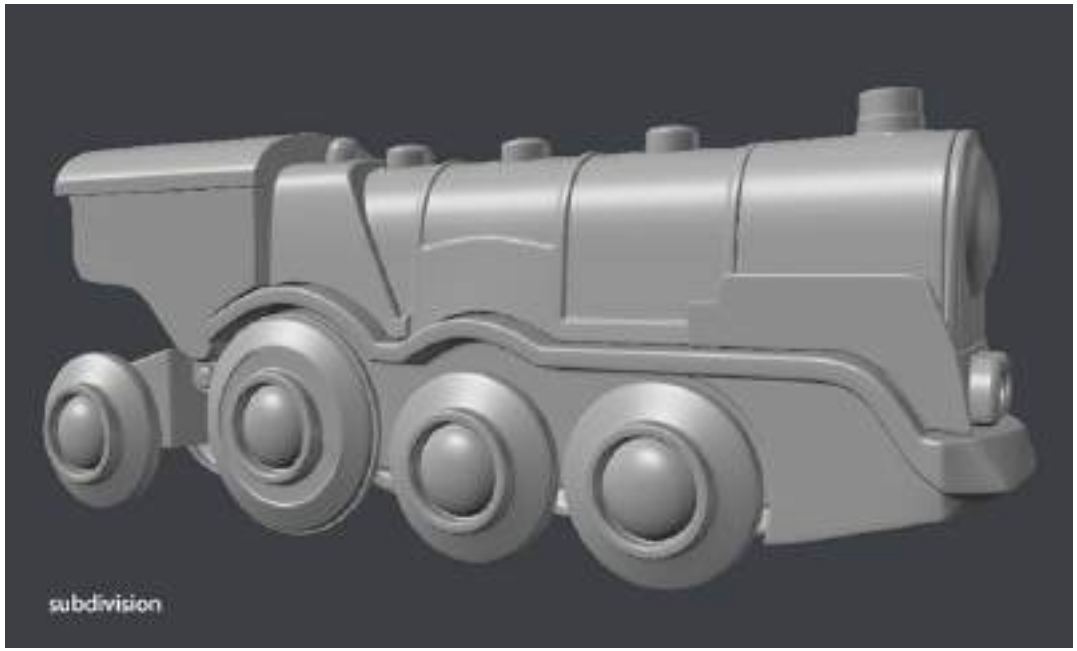
Displacement mapping

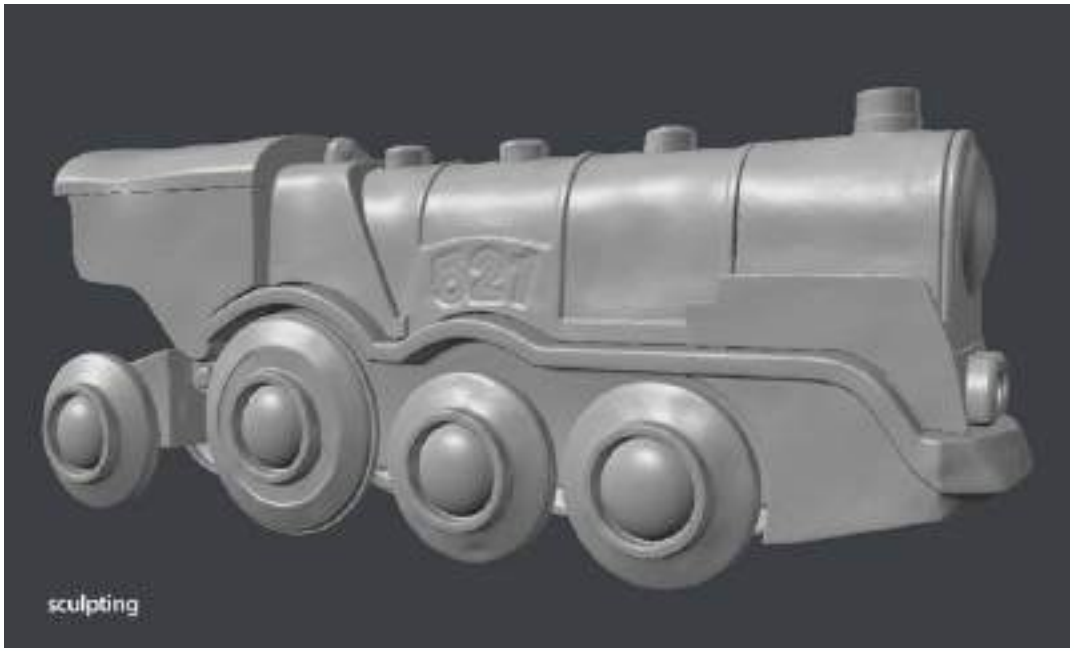
- Displacement mapping: use a texture that defines the displacement along the normal direction at each surface location



[WikimediaCommons]

© 2020 Fabio Pellacini



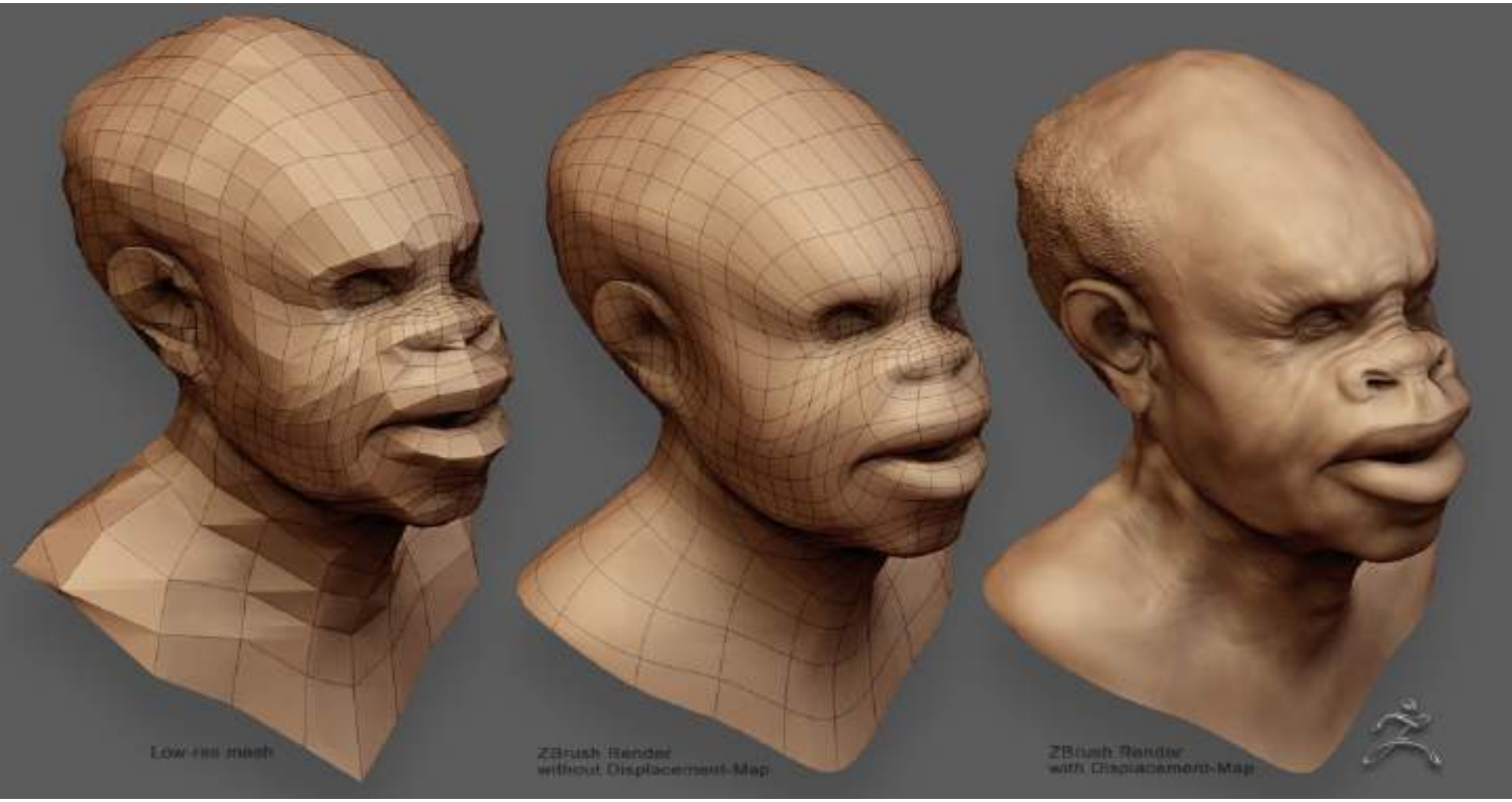


[Pawel Filip]

© 2020 Fabio Pellacini

© 2020 Fabio Pellacini

Displacement mapping



[Pixologic]

© 2020 Fabio Pellacini

Displacement mapping

[Source unknown]

© 2020 Fabio Pellacini

Displacement mapping

[Yining Karl]

© 2020 Fabio Pellacini

Displacement mapping

- Write vertex position using a parametrization to index the texture

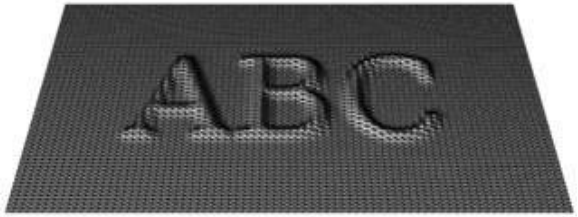
$$\mathbf{p}^d(u,v) = \mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)$$

- To render a displaced triangle mesh, we tesselate it at the appropriate resolution and displace each vertex by the amount specified in the texture along the vertex normal

$$\mathbf{p}_i^d = \mathbf{p}_i + h[u_i, v_i]\mathbf{n}_i$$

ORIGINAL MESH

DISPLACEMENT MAP

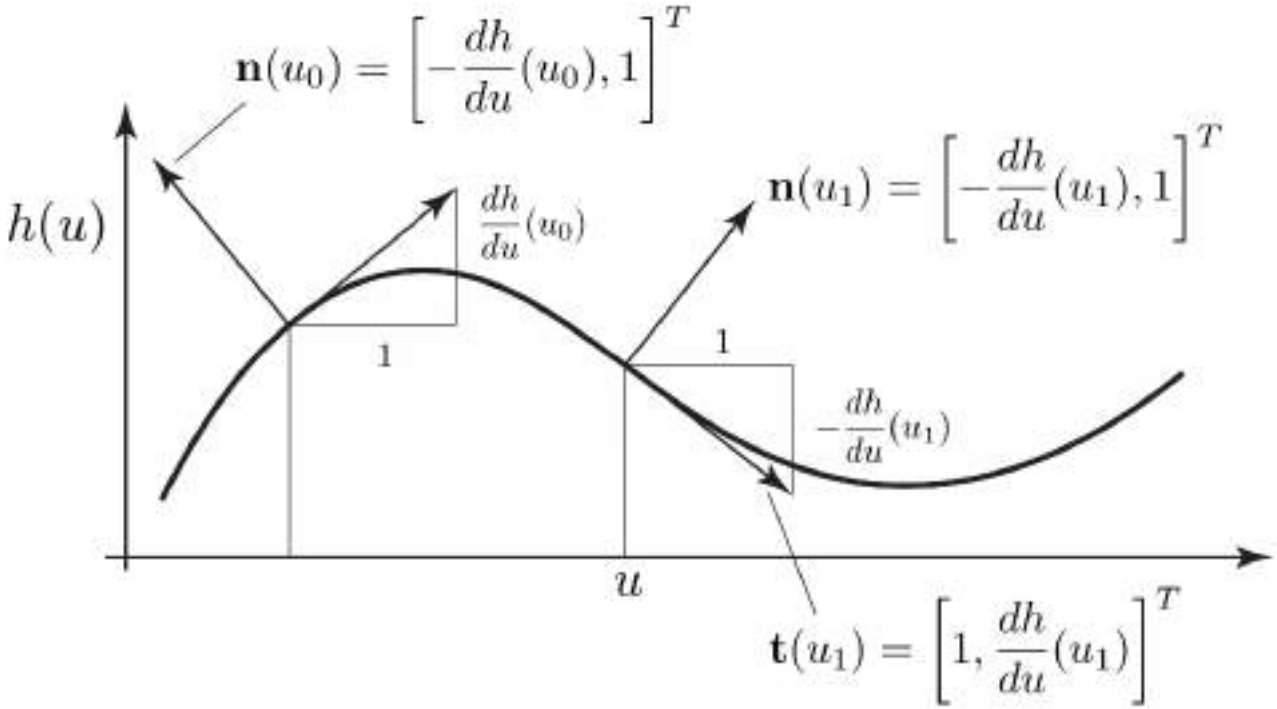


MESH WITH DISPLACEMENT

© 2020 Fabio Pellacini

Computing displaced normals

- Option 1: smooth normal algorithm ran on the displaced positions
  - sound and practical, but may be inefficient
- Option 2: estimate directly from the positions and texture



© 2020 Fabio Pellacini

Computing displaced normals

- Normals are vectors orthogonal to the tangent plane that can be computed as the cross product of two tangents
- For a continuous surface, we can write displaced normals as the cross product of the tangents of the displaced positions, which are the surface derivatives w.r.t. each parameter

$$\mathbf{n}^d(u,v) = \frac{\partial \mathbf{p}^d(u,v)}{\partial u} \times \frac{\partial \mathbf{p}^d(u,v)}{\partial v} =$$
$$= \frac{\partial \left[\mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)\right]}{\partial u} \times \frac{\partial \left[\mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)\right]}{\partial v}$$

© 2020 Fabio Pellacini

Computing displaced normals

- We can simplify the above equation by assuming that the normal changes little around the point

$$\mathbf{n}^d = \frac{\partial[\mathbf{p}+h\mathbf{n}]}{\partial u} \times \frac{\partial[\mathbf{p}+h\mathbf{n}]}{\partial v} \approx \left[\frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u}\mathbf{n}\right] \times \left[\frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v}\mathbf{n}\right] =$$

$$= \left[\frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u}\mathbf{n}\right] \times \left[\frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v}\mathbf{n}\right] = \left[\mathbf{t}_u + \frac{\partial h}{\partial u}\mathbf{n}\right] \times \left[\mathbf{t}_v + \frac{\partial h}{\partial v}\mathbf{n}\right] =$$

$$= \mathbf{t}_u \times \mathbf{t}_v + \frac{\partial h}{\partial v}(\mathbf{t}_u \times \mathbf{n}) + \frac{\partial h}{\partial v}(\mathbf{n} \times \mathbf{t}_v)$$

- For tangents that are orthogonal and normalized we have

$$\mathbf{n}^d = -\frac{\partial h}{\partial u}\mathbf{t}_u - \frac{\partial h}{\partial v}\mathbf{t}_v + \mathbf{n}$$

© 2020 Fabio Pellacini

Tangent frame

- T*angent frame:* coordinate system at each surface location, with surface normal as z and tangents as x and y

$$T = \{\mathbf{t}_u, \mathbf{t}_v, \mathbf{n}, \mathbf{p}\}$$

- Displacement can be written w.r.t. the tangent frame as

$$\mathbf{p}^d = T \begin{bmatrix} 0 & 0 & h & 1 \end{bmatrix}^T \quad \mathbf{n}^d = T \begin{bmatrix} -\frac{\partial h}{\partial u} & -\frac{\partial h}{\partial v} & 1 & 0 \end{bmatrix}^T$$

- Can think of the tangent frame as defining a plane that locally approximates the surface
- Tangent frame depends on texture orientation since tangents are defined w.r.t. *(u,v)*

© 2020 Fabio Pellacini

Tangent frame for meshes

- On triangle meshes, derive tangent frames from vertex positions and texture coordinates
  - derivation in www.terathon.com/code/tangent.html
- Would like to define two tangents ($\mathbf{t}_u$, $\mathbf{t}_v$) such as we can write all points $\mathbf{p}$ on a triangle in terms of the triangle *(u,v)* as

$$\mathbf{p}(u,v) = \mathbf{v}_0 + (u - u_0)\mathbf{t}_u + (v - v_0)\mathbf{t}_v$$

- We impose two constraints on the other triangle vertices

$$\begin{cases} \mathbf{p}(u_1, v_1) = \mathbf{v}_1 \\ \mathbf{p}(u_2, v_2) = \mathbf{v}_2 \end{cases} \Rightarrow \begin{cases} \mathbf{v}_1 = \mathbf{v}_0 + (u_1 - u_0)\mathbf{t}_u + (v_1 - v_0)\mathbf{t}_v \\ \mathbf{v}_2 = \mathbf{v}_0 + (u_2 - u_0)\mathbf{t}_u + (v_2 - v_0)\mathbf{t}_v \end{cases}$$

© 2020 Fabio Pellacini

Tangent frames for meshes

- We can manipulate a bit the previous equations as

$$\begin{cases} \mathbf{v}_1 - \mathbf{v}_0 = (u_1 - u_0)\mathbf{t}_u + (v_1 - v_0)\mathbf{t}_v \\ \mathbf{v}_2 - \mathbf{v}_0 = (u_2 - u_0)\mathbf{t}_u + (v_2 - v_0)\mathbf{t}_v \end{cases}$$

- For notation we can rewrite it with easier constants as

$$\begin{cases} \mathbf{e}_1 = s_1\mathbf{t}_u + t_1\mathbf{t}_v \\ \mathbf{e}_2 = s_2\mathbf{t}_u + t_2\mathbf{t}_v \end{cases}$$

- This is a linear system of six equations in six unknowns ($\mathbf{t}_u$, $\mathbf{t}_v$)
- We can rewrite it again in matrix notation as

$$\begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{t}_u & \mathbf{t}_v \end{bmatrix} \begin{bmatrix} s_1 & s_2 \\ t_1 & t_2 \end{bmatrix}$$

© 2020 Fabio Pellacini

Tangent frames for meshes

- We now multiply both sides by the inverse of the *(s,t)* matrix

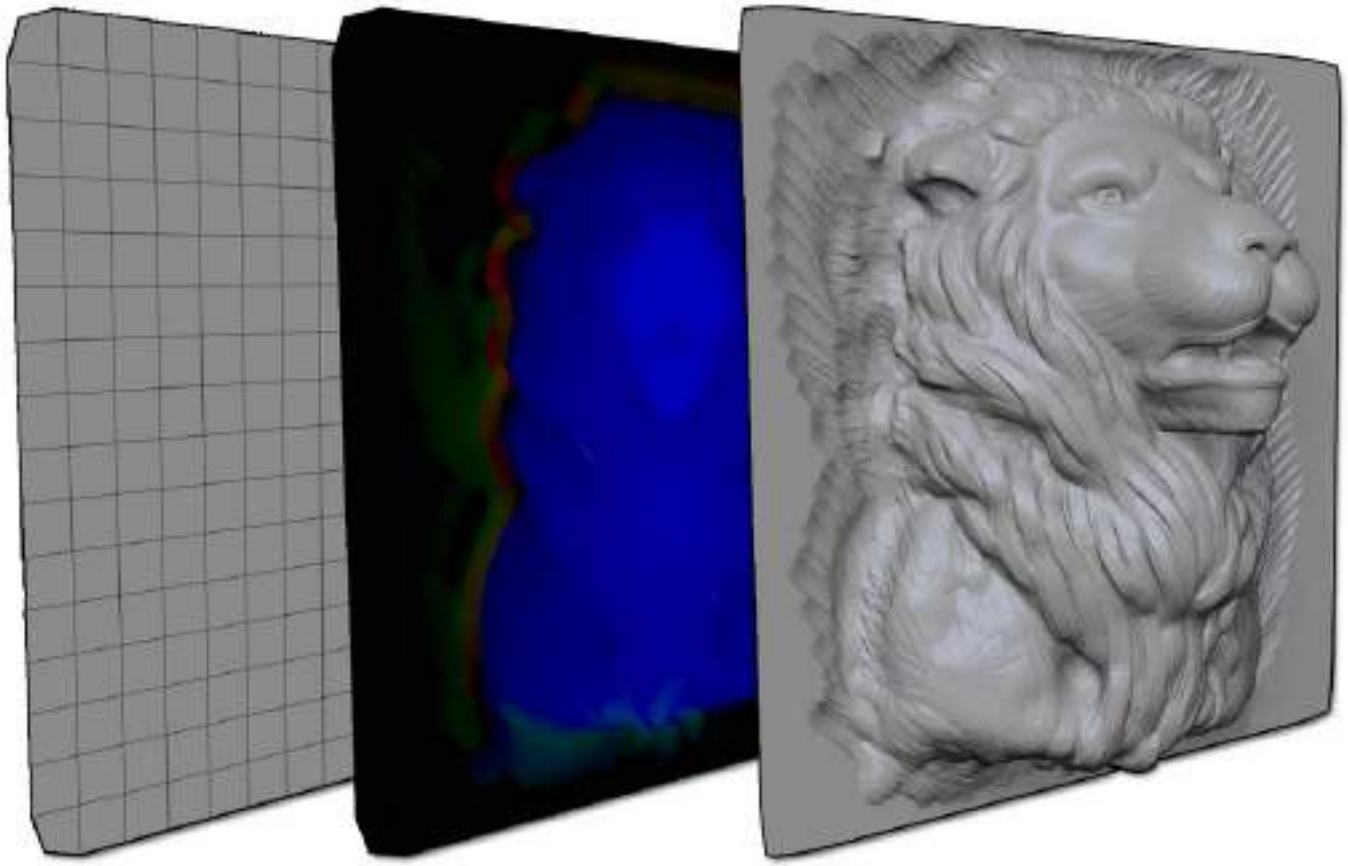
$$\begin{bmatrix}\mathbf{t}_u & \mathbf{t}_v\end{bmatrix} = \begin{bmatrix}\mathbf{e}_1 & \mathbf{e}_2\end{bmatrix}\left(\frac{1}{s_1 t_2 - s_2 t_1}\begin{bmatrix} t_2 & -s_2 \\ -t_1 & s_1 \end{bmatrix}\right)$$

- This gives us the two required tangents for each triangle
- To get tangents, for the whole mesh, we compute smooth vertex tangents in the same manner we did to compute vertex normals
  - note that the tangents are not orthonormal, but we can address this with orthonormalization as written before
  - the most common way to store the tangents is to only store one of them per-vertex and the sign of their cross product
  - we then recuperate the other tangent with the cross product with the normal

© 2020 Fabio Pellacini

Vector displacement

- Extend displacement mapping by specifying a tangent space vector as the offset for the new position



[Pixologic]

© 2020 Fabio Pellacini

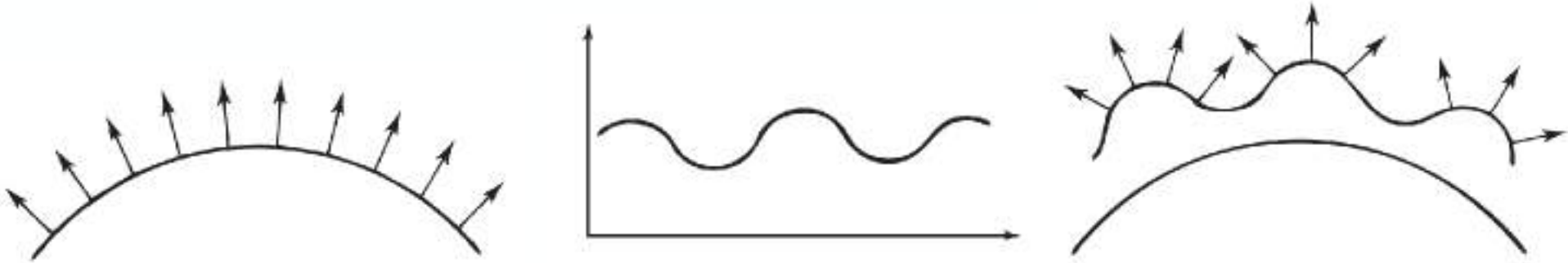
Vector displacement

[Yining Karl]

© 2020 Fabio Pellacini

Bump mapping

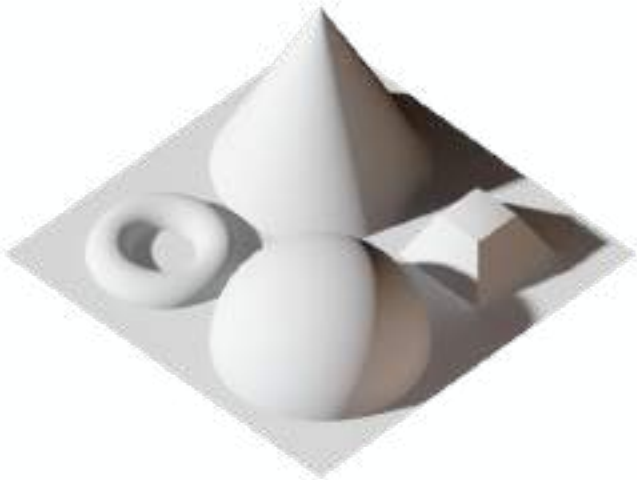
- Displacement mapping is expensive since it requires tessellation
- For small displacement, most effects are in the normal
- *Bump mapping*: us displaced normal for shading, but original geometry for intersection and rasterization

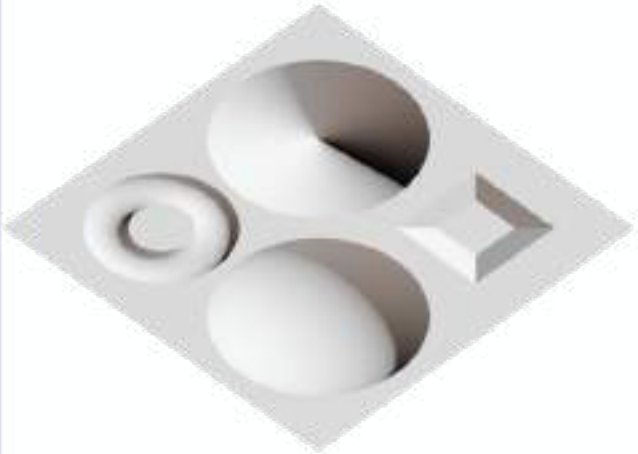


(a) original surface (b) height map (d) perturbed normals

Normal mapping

- *Normal mapping*: directly store normals as texture colors
  - encode with (r,g,b) = 0.5 (x,y,z) + 0.5
  - decode with (x,y,z) = 2 (r,g,b) - 1



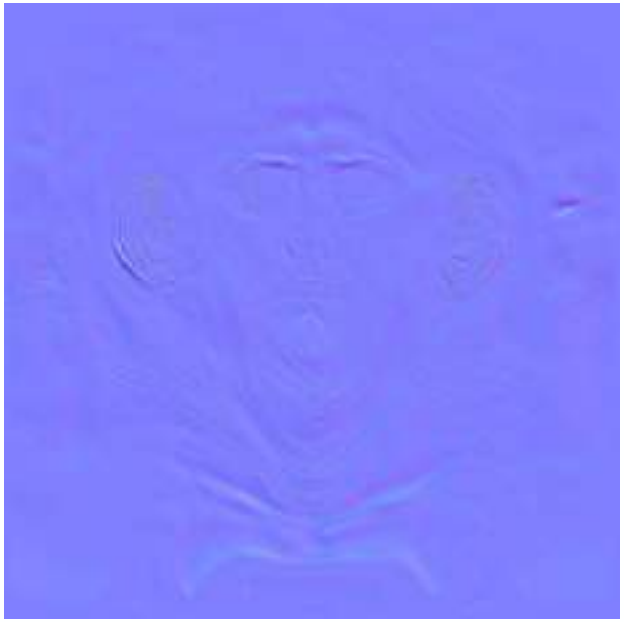


[WikimediaCommons]

© 2020 Fabio Pellacini

Normal mapping

- More general than bump mapping: can store arbitrary normals
- Mostly used in industry since it supports simplification of meshes



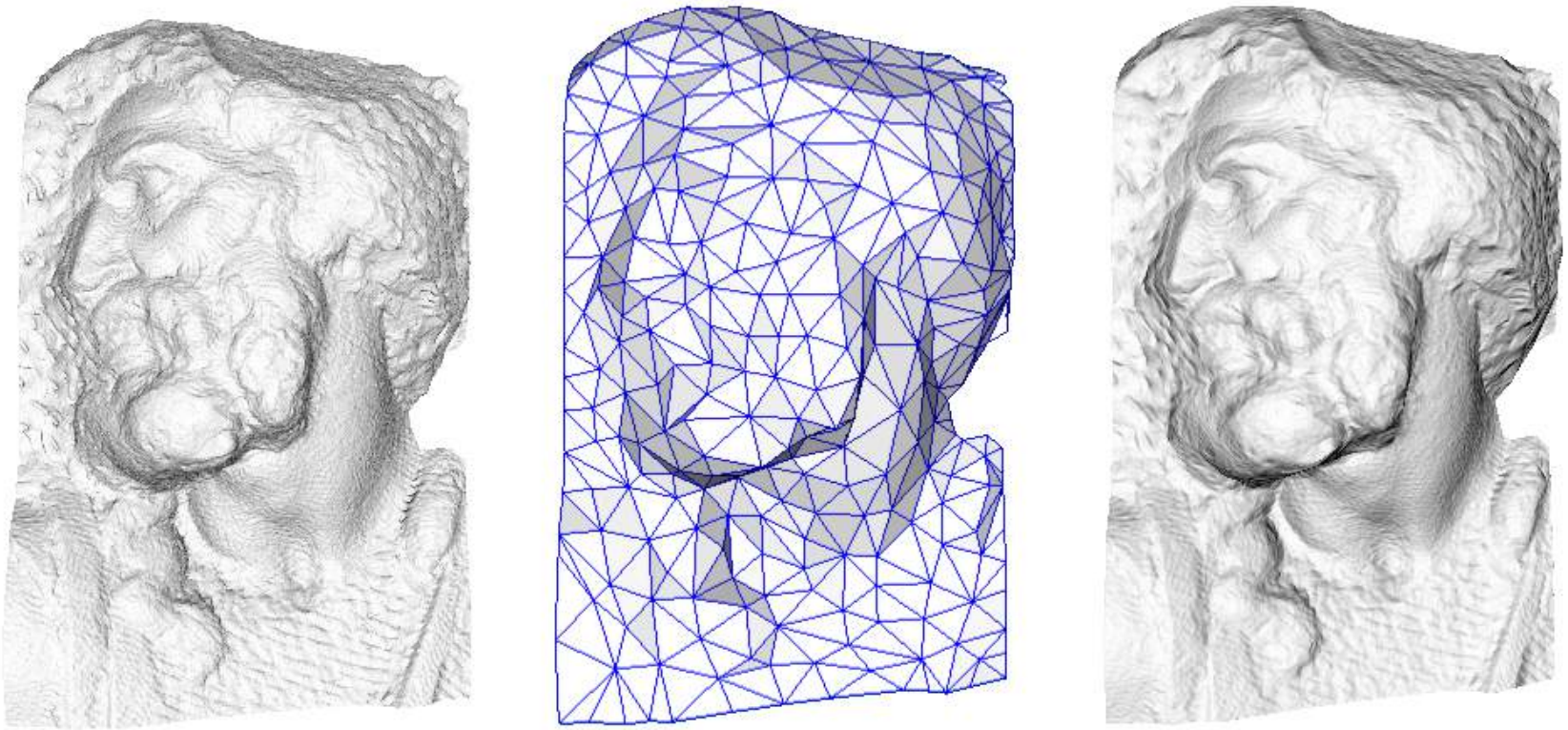
[BlenderNerd]

[Lee Perry-Smith]

© 2020 Fabio Pellacini

Normal mapping

- Mostly used in industry since it supports simplification of meshes



original mesh
4M triangles

simplified mesh
500 triangles

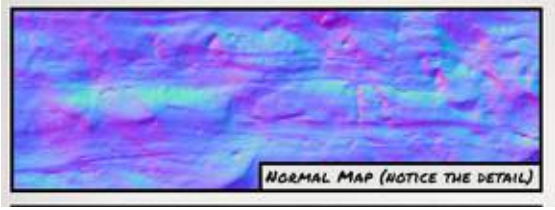
simplified mesh
and normal mapping
500 triangles

[Paolo Cignoni]

© 2020 Fabio Pellacini

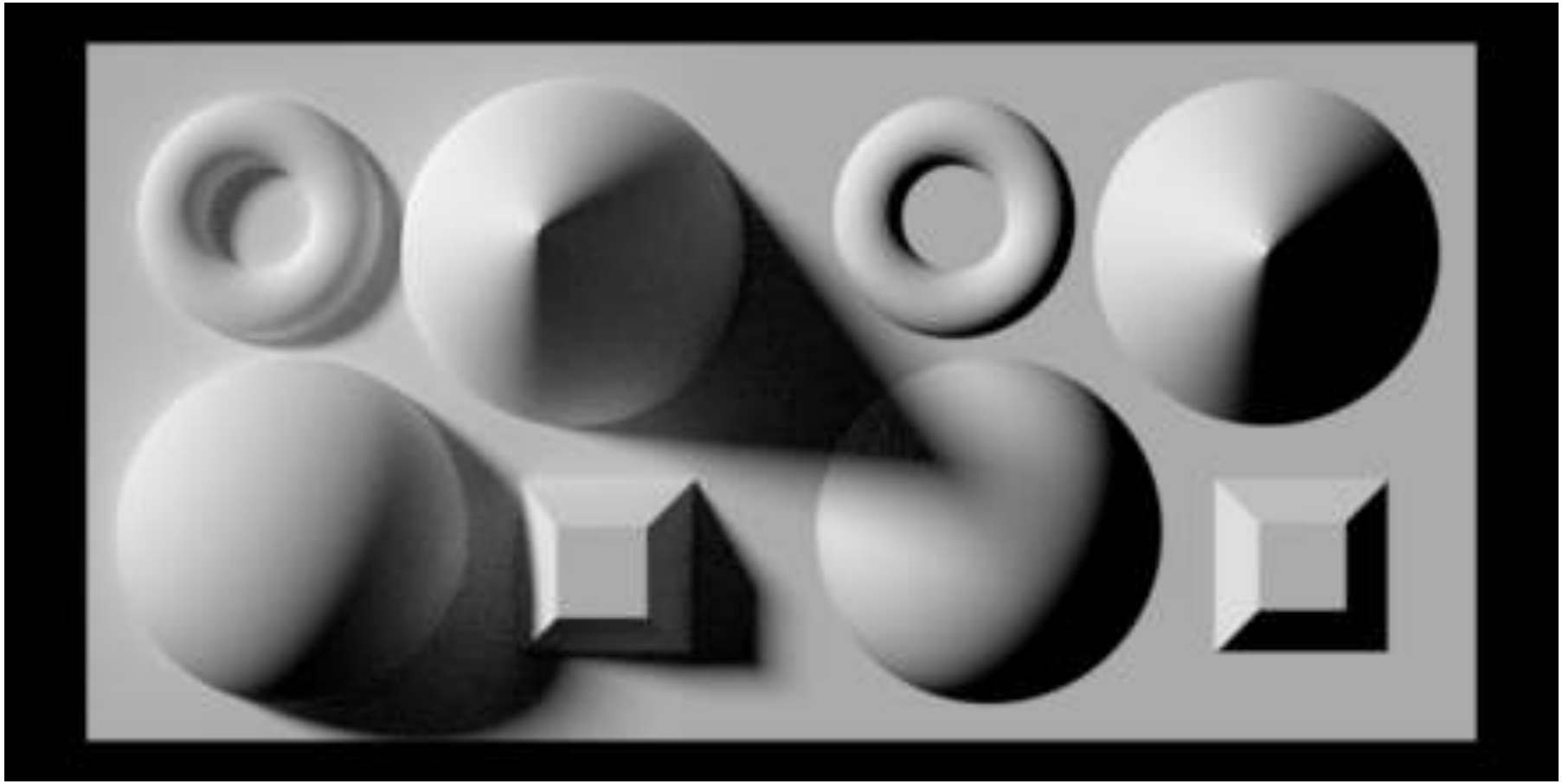
Bump vs normal vs displacement

[Andrew Letourneau]



© 2020 Fabio Pellacini

Bump vs normal vs displacement



[WikimediaCommons]

© 2020 Fabio Pellacini

Normal and displacement

- Entertainment apps use normal or displacement everywhere
- Normal maps: used for real-time applications
  - mostly from compressed meshes
- Displacement maps: used in offline rendering for movies
  - production renderers are optimized for displacement
  - handle high details, while the rest of the pipeline uses "low-poly" control vertices

© 2020 Fabio Pellacini

Editing Surfaces

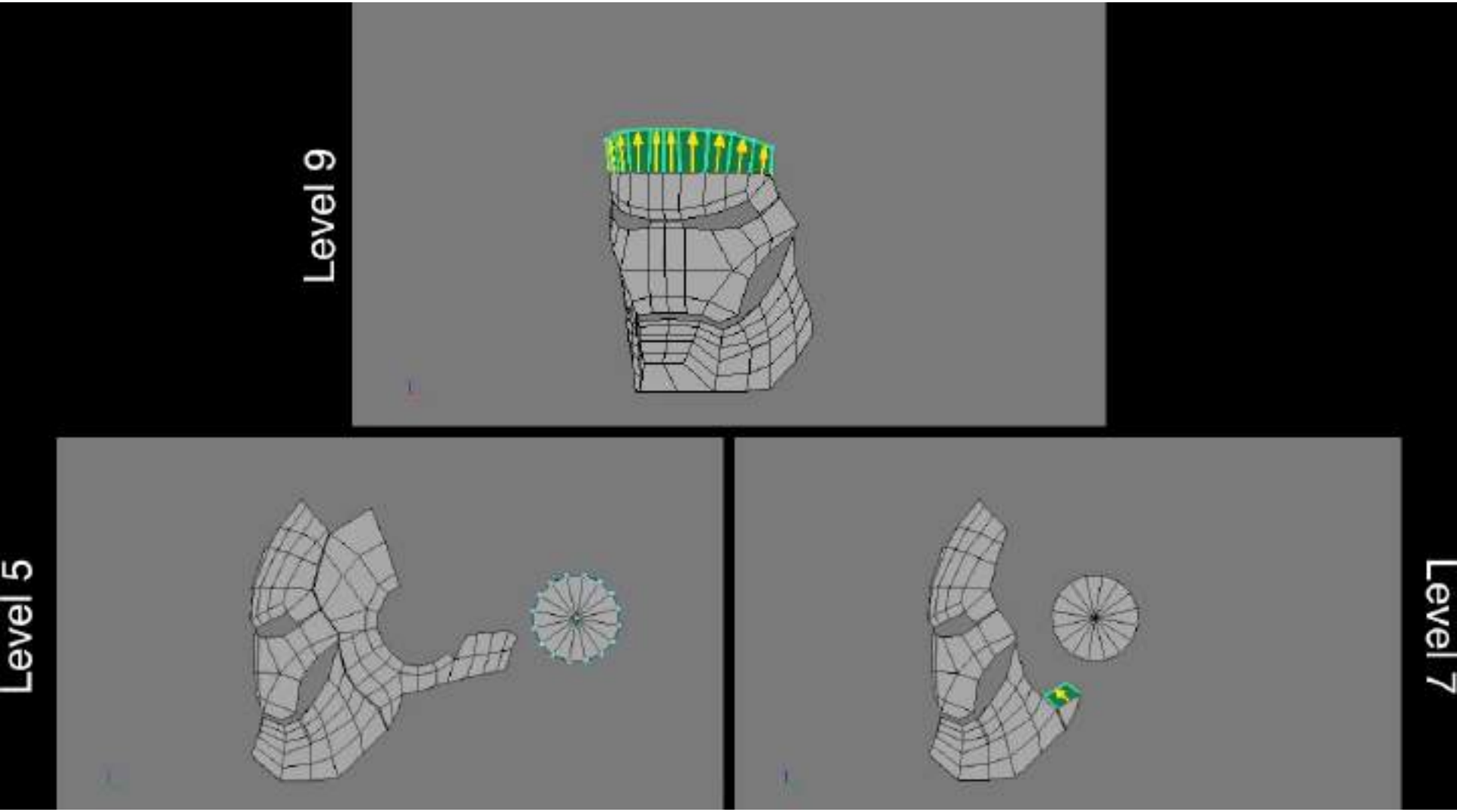
© 2020 Fabio Pellacini

Two different workflows

- Low-poly/subdiv modeling
  - topology and geometric operations
  - carefully modify each vertex
  - care a lot about mesh topology
  - hard to use but precise and efficient surfaces
- High-poly sculpting
  - sculpt like clay using "brushes"
  - mostly care about geometry, very few topology changes
  - easy to use but imprecise and highly tessellated surfaces
  - "retopology" tools to convert sculpt to subdiv

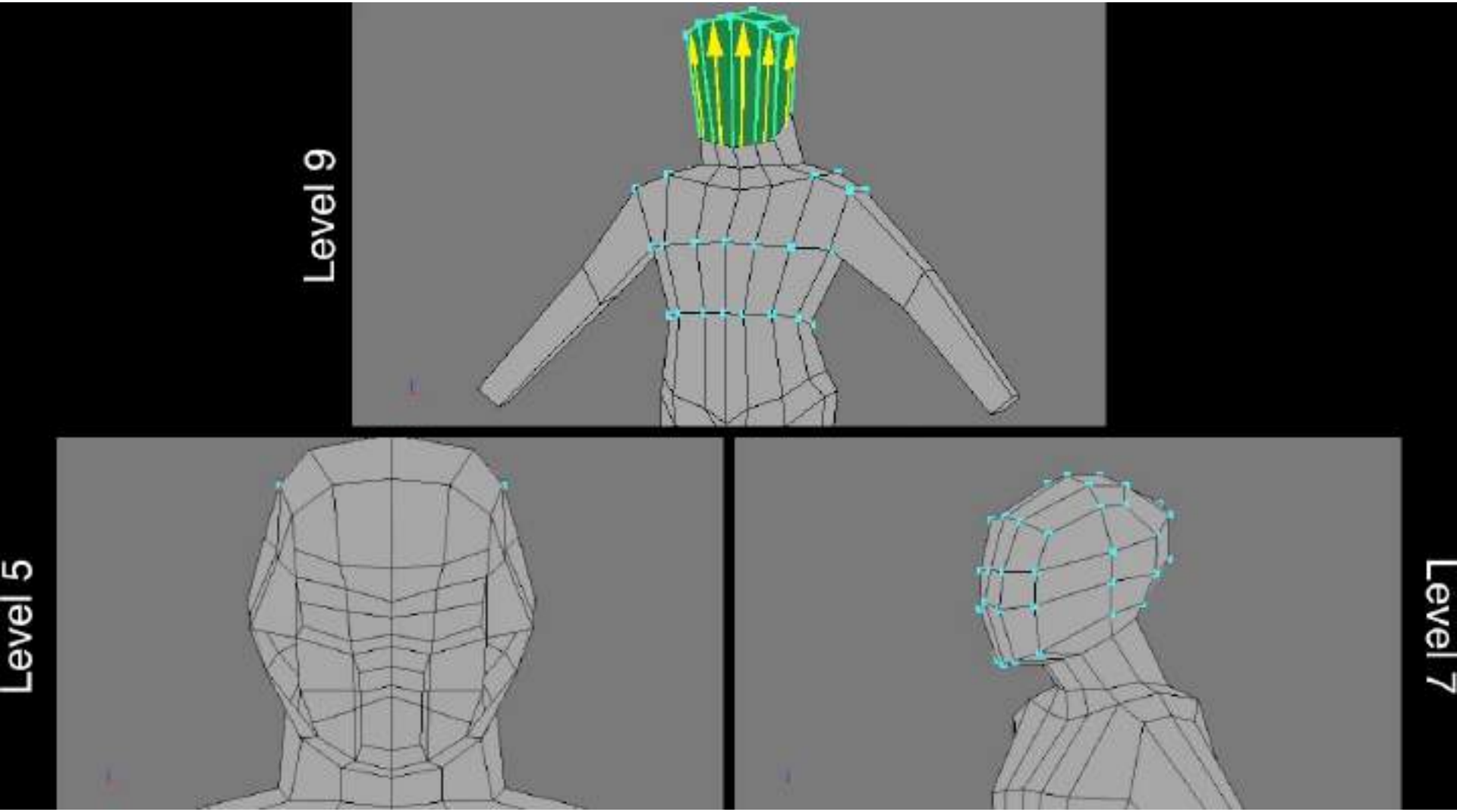
© 2020 Fabio Pellacini

Low poly modeling



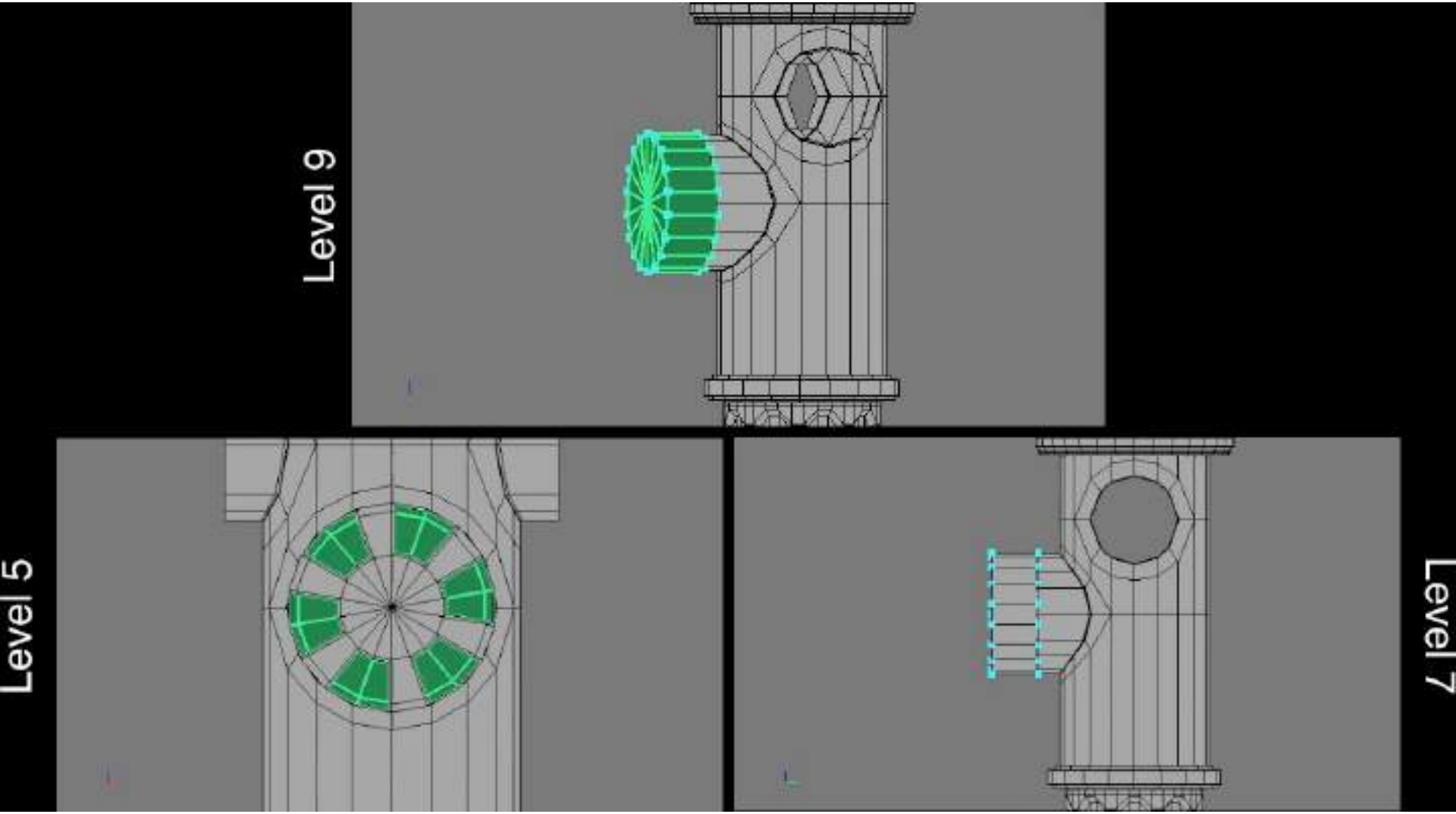
© 2020 Fabio Pellacini

Low poly modeling



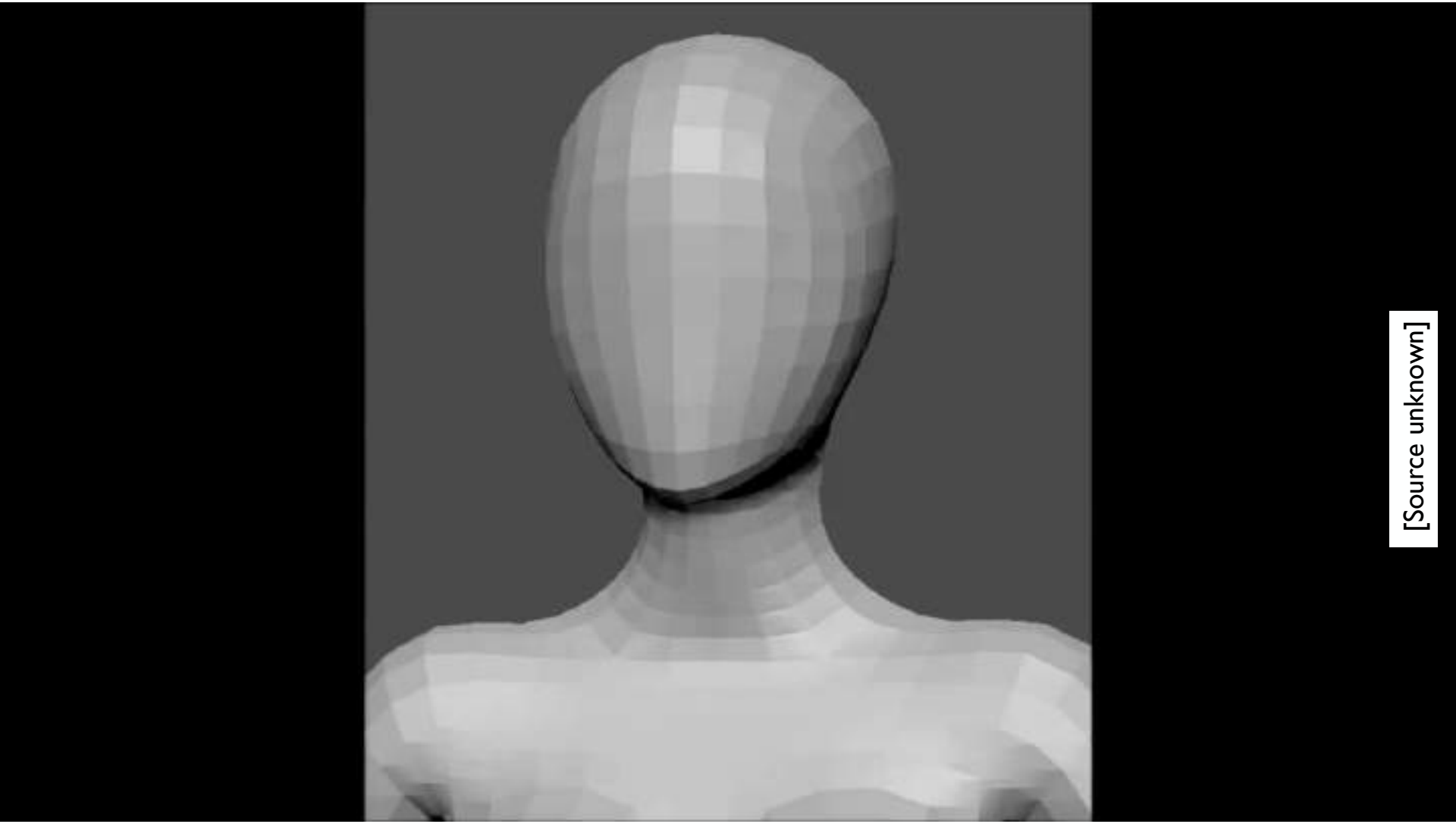
© 2020 Fabio Pellacini

Low poly modeling



© 2020 Fabio Pellacini

Sculpting



[Source unknown]

© 2020 Fabio Pellacini

Sculpting

© 2020 Fabio Pellacini

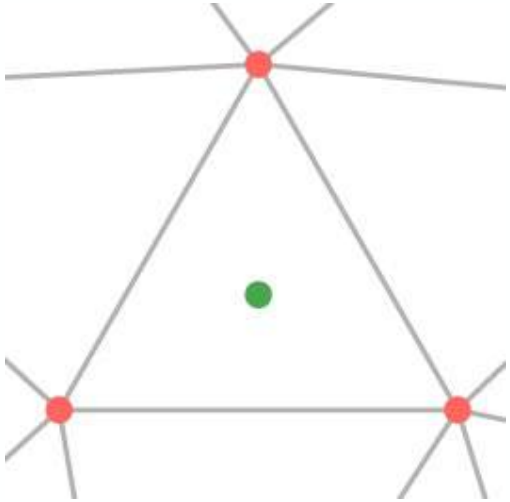
Triangle Mesh Data Structures

© 2020 Fabio Pellacini

Separate triangles

- Store each triangle independently

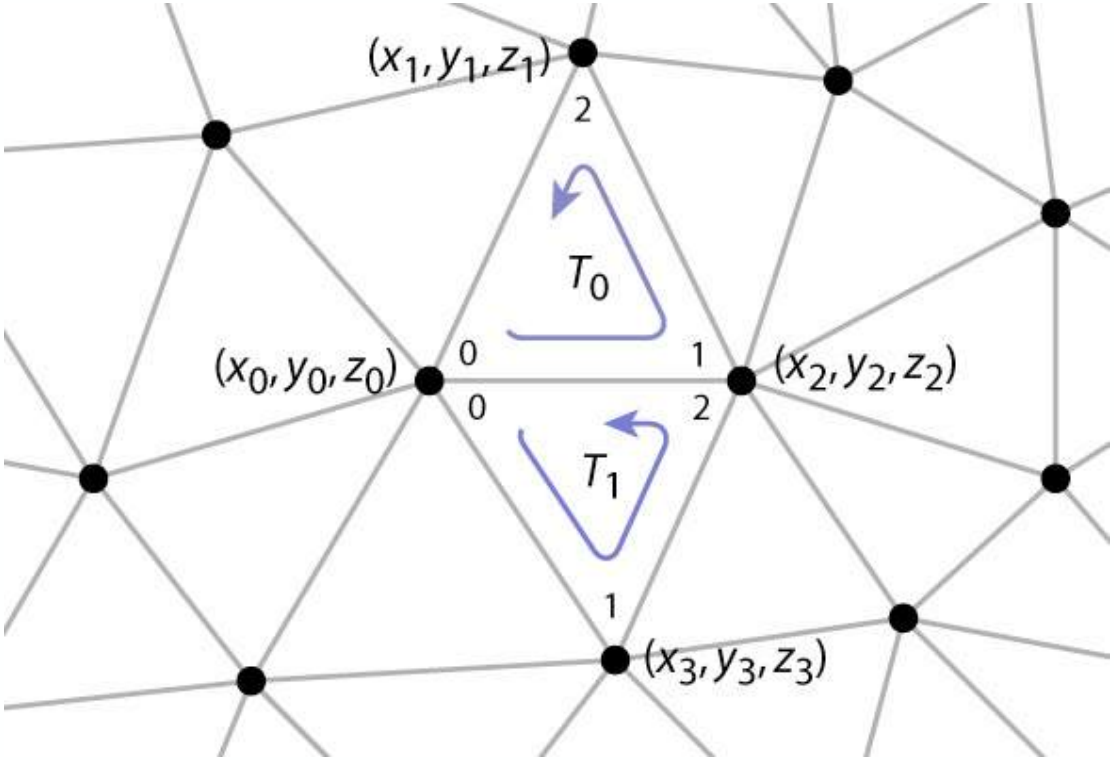
```
struct Mesh {
  vec3f pos[nT][3];
};
```



© 2020 Fabio Pellacini

Separate triangles

	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮



© 2020 Fabio Pellacini

Separate triangles

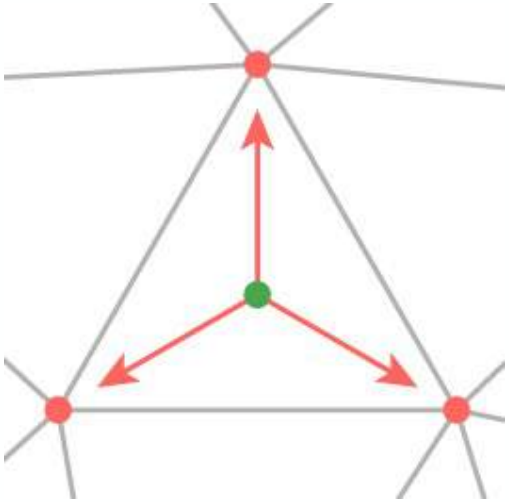
- Array of triples of points
  - float[n_T][3][3]: about 72 bytes per vertex
  - 2 triangles per vertex (on average)
- Various problems
  - wastes space (each vertex stored 6 times)
  - cracks due to roundoff
  - difficulty of finding neighbors at all

© 2020 Fabio Pellacini

Indexed triangles

- Store each vertex once
- Each triangle points to its three vertices

```
struct Mesh {
  vec3i tris[nT];
  vec3f pos[nV];
};
```

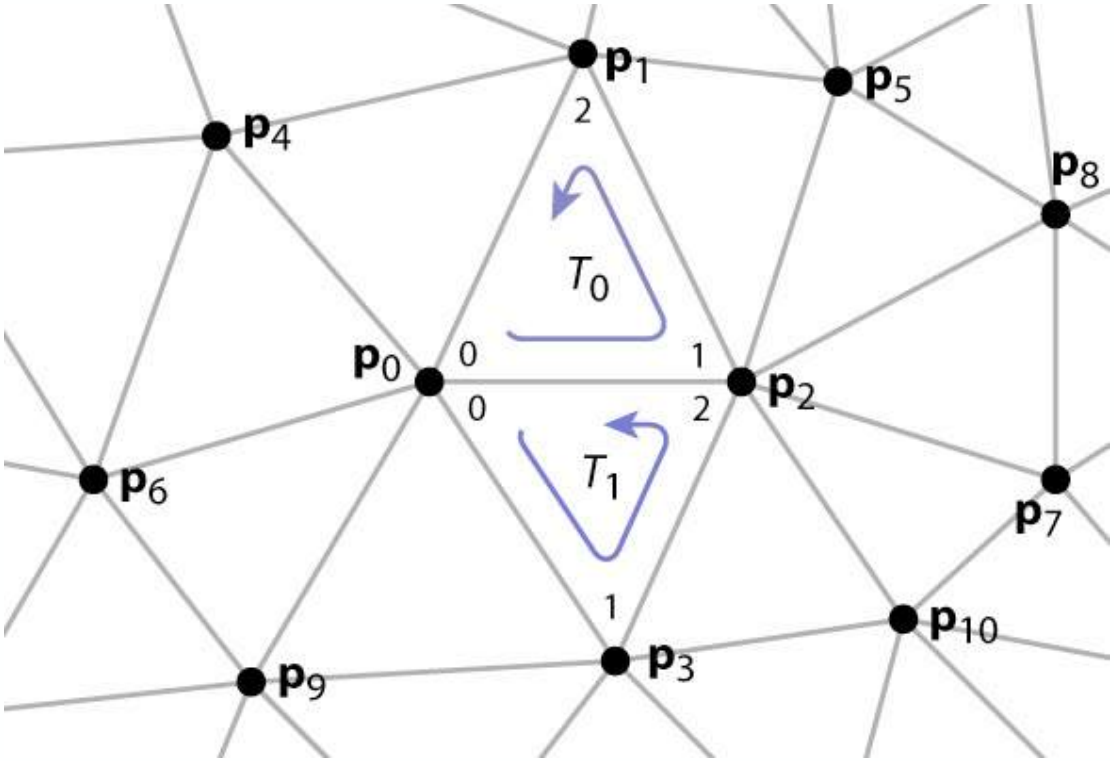


© 2020 Fabio Pellacini

Indexed triangles

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
	⋮



© 2020 Fabio Pellacini

Indexed triangles

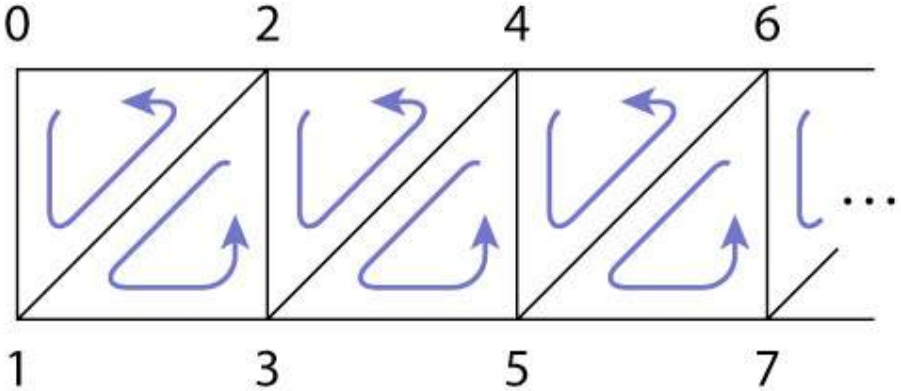
- Array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
- Array of triangle indices
  - int[n_T][3]: 12 bytes per triangle
  - 2 triangles per vertex on average: 24 bytes per vertex
- Total storage: 36 bytes per vertex (factor of 2 savings)
- Represents topology and geometry separately
- Finding neighbors is well defined

© 2020 Fabio Pellacini

Triangle strips

- Compress triangle lists by skipping repeated indices
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing the second and third vertex of the previous triangle
  - for long strips, this requires about one index per triangle

```
struct Mesh {
  int strips[nT][];
  float pos[nV][3];
};
```

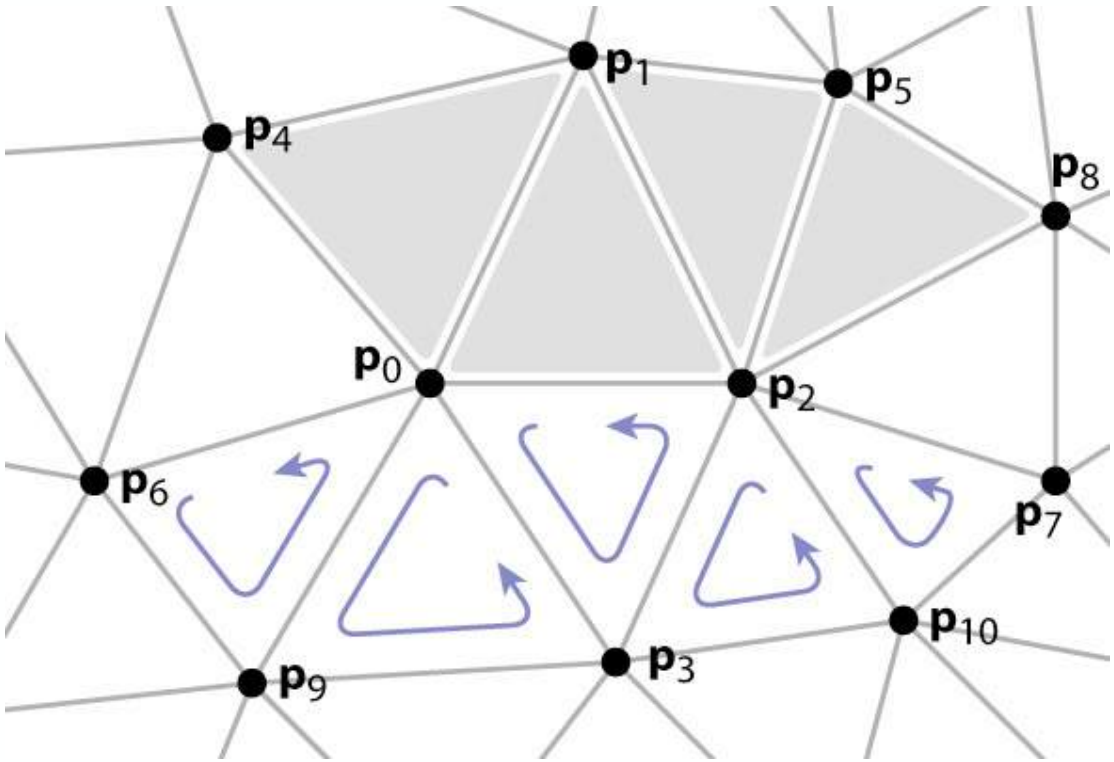


© 2020 Fabio Pellacini

Triangle strips

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tStrip[0]	6, 0, 4, 1, 2, 5, 8
tStrip[1]	6, 9, 0, 3, 2, 10, 7
	⋮



© 2020 Fabio Pellacini

Triangle strips

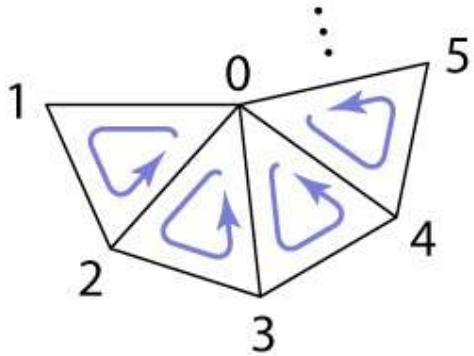
- Array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
- Array of index lists
  - int[n_S][variable]: (2 + n)x4 bytes per strip
  - on average, (1 + ε) indices per triangle (assuming long strips)
- Total is 20 bytes per vertex (limiting best case)
  - factor of 3.6 over separate triangles; 1.8 over indexed mesh

© 2020 Fabio Pellacini

Triangle fans

- Same idea as triangle strips, but keep oldest rather than newest
  - every sequence of three vertices produces a triangle
  - for long fans, this requires about one index per triangle
- Memory considerations exactly the same as triangle strip

```
struct Mesh {
  int fans[nT][];
  float pos[nV][3];
};
```



© 2020 Fabio Pellacini

Data on meshes

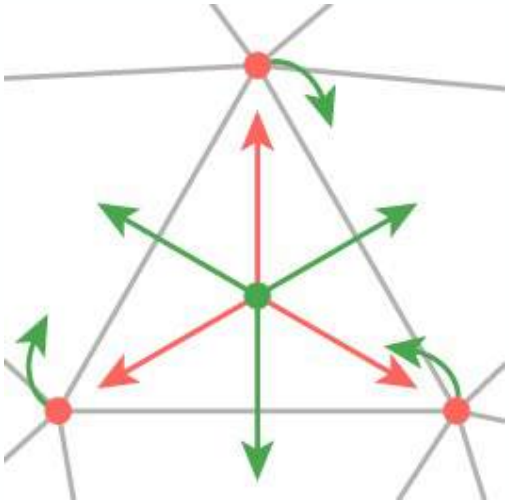
- Often need to store additional information besides just the geometry
- Can store additional data at faces, vertices, or edges
- Store on faces for discontinuous data, e.g. facet colors
- Information about creases stored at edges
- Quantities that vary continuously (without sudden changes, or discontinuities) gets stored at vertices

© 2020 Fabio Pellacini

Triangle neighbor structure

- Extension to indexed triangle set
- Triangle points to its three neighboring triangles
- Vertex points to a single neighboring triangle
- Can now enumerate triangles around a vertex

```
struct Mesh {
  float pos[nV][3];
  int tris[nT][3];
  int tAdj[nT][3];
  int vAdj[nV];
};
```



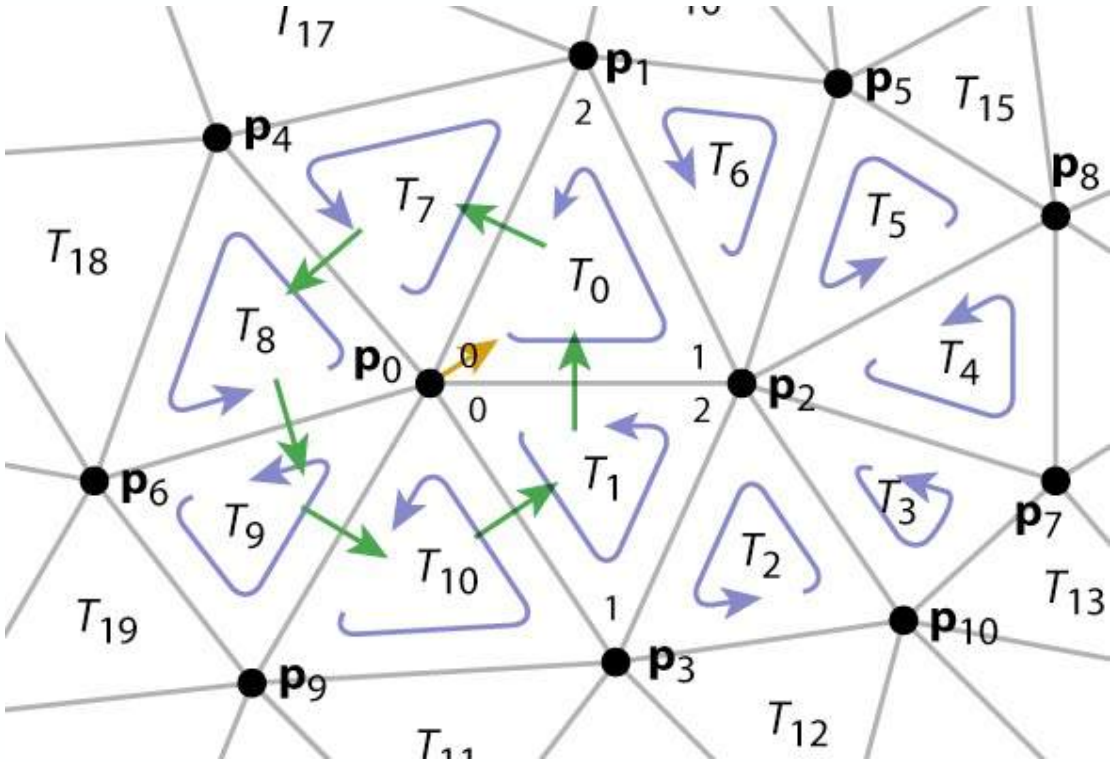
© 2020 Fabio Pellacini

Triangle neighbor structure

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
tInd[2]	10, 2, 3
tInd[3]	2,10,7
	⋮

tNbr[0]	1, 6, 7
tNbr[1]	10, 2, 0
tNbr[2]	3, 1, 12
tNbr[3]	2, 13, 4
	⋮

vTri[0]	0
vTri[1]	6
vTri[2]	1
vTri[3]	1
	⋮



© 2020 Fabio Pellacini

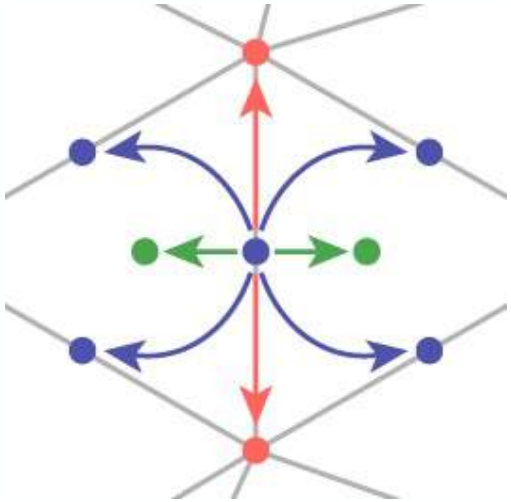
Triangle neighbor structure

- Indexed mesh was 36 bytes per vertex
- Add an array of triples of indices (per triangle)
  - int[n_T][3]: about 24 bytes per vertex
- Add an array of representative triangle per vertex
  - int[n_V]: 4 bytes per vertex
- Total storage: 64 bytes per vertex
  - still not as much as separate triangles

© 2020 Fabio Pellacini

Winged-edge mesh

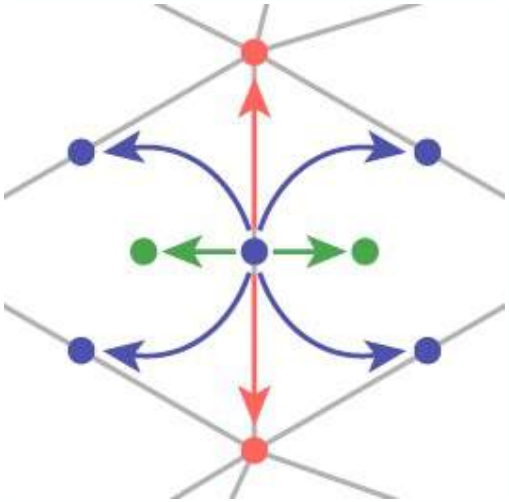
- Edge-centric rather than face-centric (works for general polygons)
- Each (oriented) edge points to:
  - left and right forward edges, left and right backward edges
  - front and back vertices, left and right faces
- Each face or vertex points to one edge



© 2020 Fabio Pellacini

Winged-edge mesh

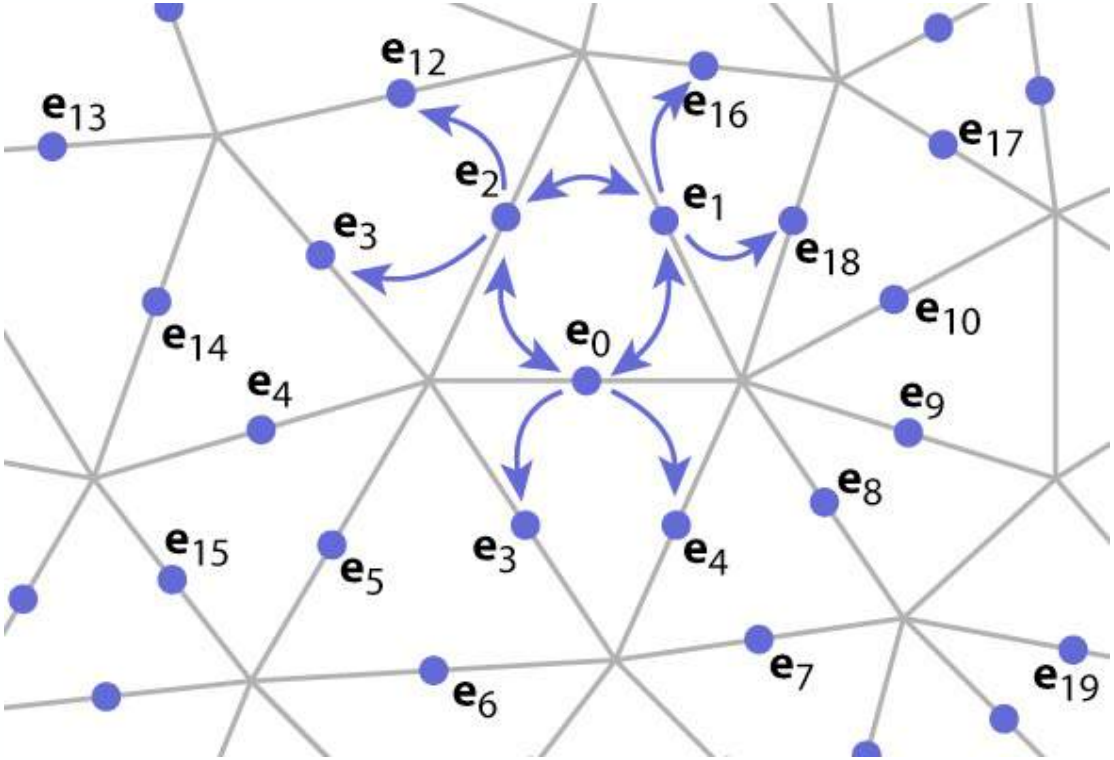
```
struct Edge {
  // edges
  int hl, hr, tl, tr;
  int h, t; // vertices
  int l, r; // face
};

struct Mesh {
  Edge edges[nE];
  int fEdge[nT];
  // vertex props…
};
```



© 2020 Fabio Pellacini

Winged-edge structure

	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
	⋮			



© 2020 Fabio Pellacini

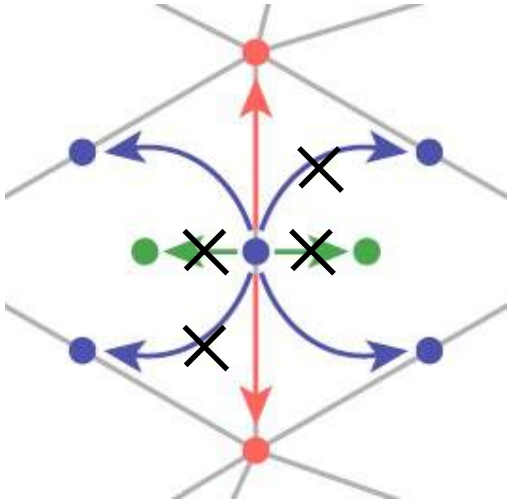
Winged-edge structure

- Array of vertex positions: 12 bytes/vert
- Array of 8-tuples of indices (per edge)
  - head/tail left/right edges + head/tail verts + left/right tris
  - int[n_E][8]: about 96 bytes per vertex
- Add a representative edge per vertex
  - int[n_V]: 4 bytes per vertex
- Total storage: 112 bytes per vertex
  - but it is cleaner and generalizes to polygon meshes

© 2020 Fabio Pellacini

Winged-edge optimizations

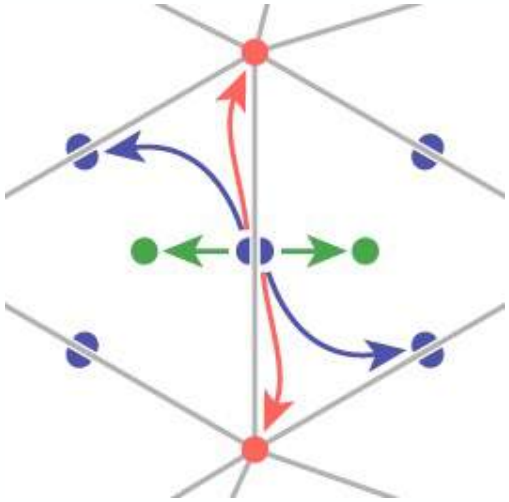
- Omit faces if not needed
- Omit one edge pointer on each side
  - results in one-way traversal



© 2020 Fabio Pellacini

Half-edge structure

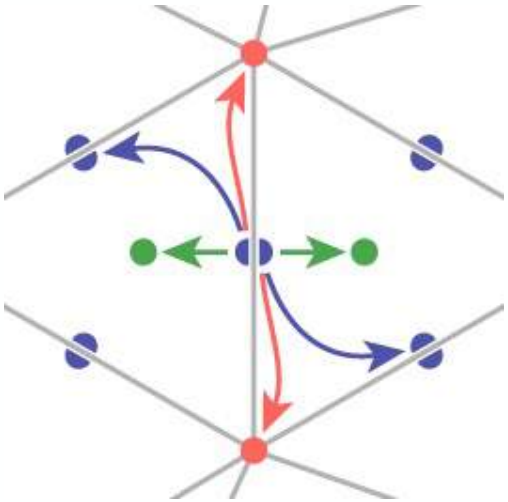
- Simplifies winged edge but still works for polygon meshes
- Each half-edge points to next edge (left forward), next vertex (front), the face (left) and the opposite half-edge
- Each face or vertex points to one half-edge



© 2020 Fabio Pellacini

Half-edge structure

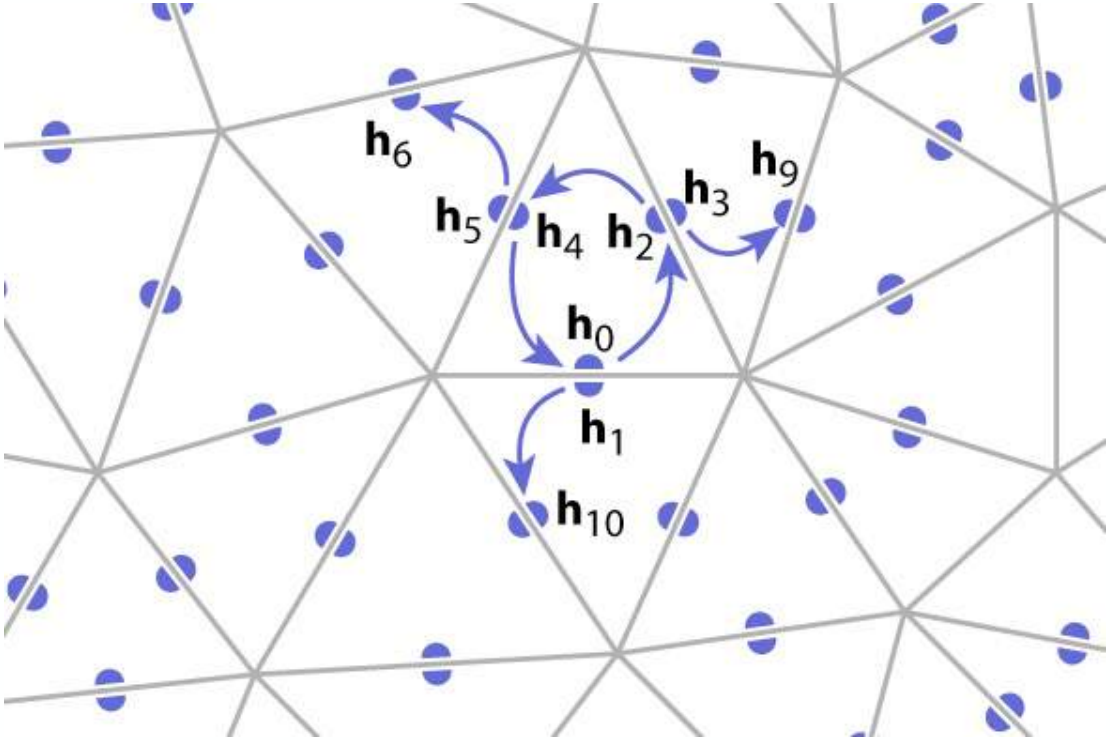
```
struct HalfEdge {
  int next;
  int opposing;
  int face;
  int vertex;
};

struct Mesh {
  HalfEdge edges[nE];
  int fEdge[nT];
  // vertex props…
};
```



© 2020 Fabio Pellacini

Half-edge structure

	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



© 2020 Fabio Pellacini

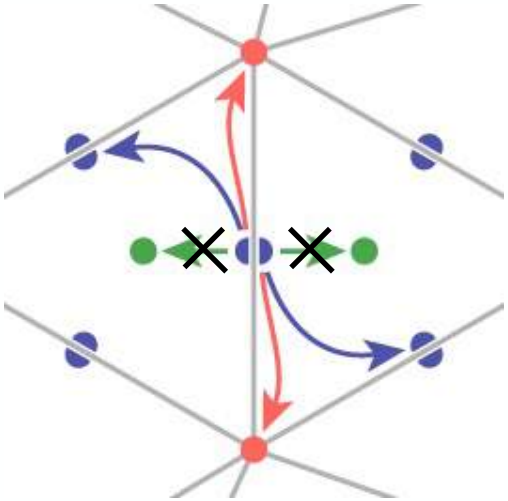
Half-edge structure

- Array of vertex positions: 12 bytes/vert
- Array of 4-tuples of indices (per h-edge)
  - next, pair h-edges + head vert + left tri
  - int[$2n_E$][4]: about 96 bytes per vertex
    - 6 h-edges per vertex (on average)
    - (4 indices x 4 bytes) per h-edge
- Add a representative h-edge per vertex
  - int[n_V]: 4 bytes per vertex
- Total storage: 112 bytes per vertex

© 2020 Fabio Pellacini

Half-edge optimizations

- Omit faces if not needed
- Use implicit pair pointers
  - they are allocated in pairs
  - they are even and odd in an array



© 2020 Fabio Pellacini

Data Structure in Practices

- Most used is the half-edge, but really rare compared to indexed
- Only useful for very specific problems and very hard to use
  - when not needed, performance is often worst then indexed
- In our work, we rebuild adjacency tables if needed

© 2020 Fabio Pellacini

Ray Tracing

Prof. Fabio Pellacini
[some original slides by Prof. Steve Marschner]

© 2020 Fabio Pellacini •

Approaches to rendering

- **Rasterization**: project each object onto image
  - real-time rendering and GPUs

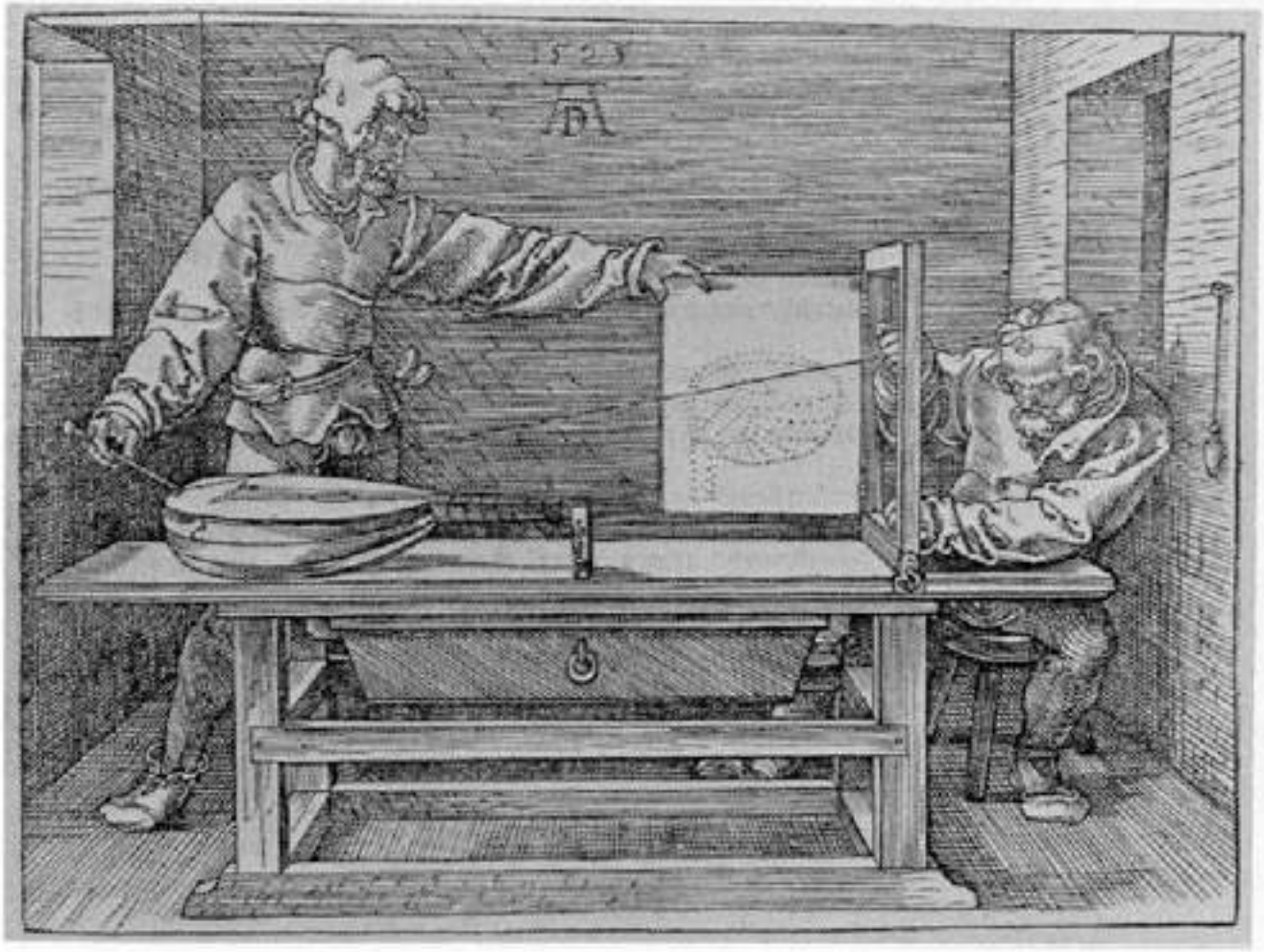
```
foreach object in scene:
  foreach pixel in image:
    if object affects pixel:
      do_something()
```

- **Raytracing**: project each pixel onto the objects
  - offline rendering, realistic rendering

```
foreach pixel in image:
  foreach object in scene:
    if object affects pixel:
      do_something()
```


© 2020 Fabio Pellacini

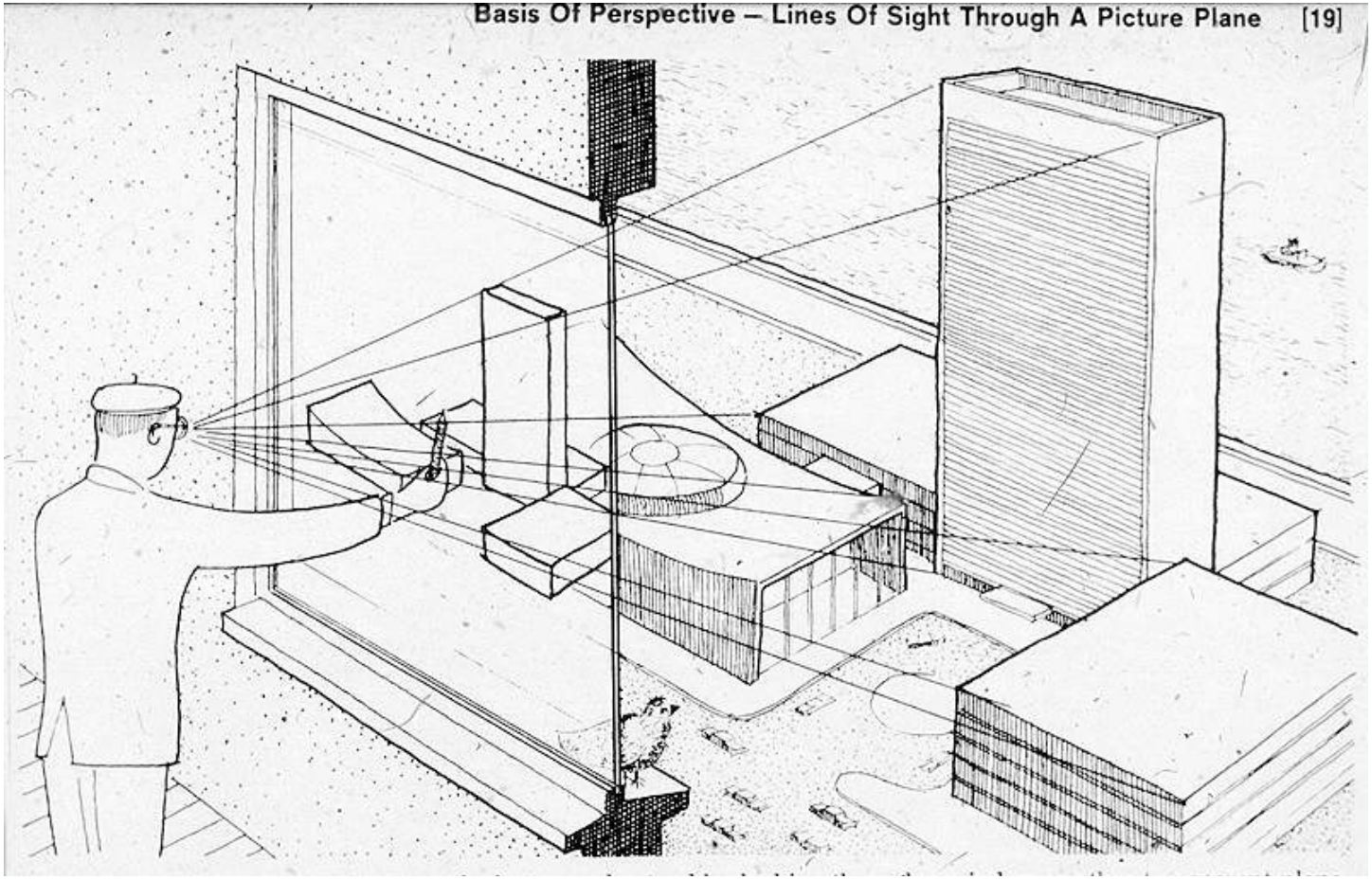
Idea Behind Raytracing



[Albrecht Dürer]

© 2020 Fabio Pellacini •

Idea Behind Raytracing

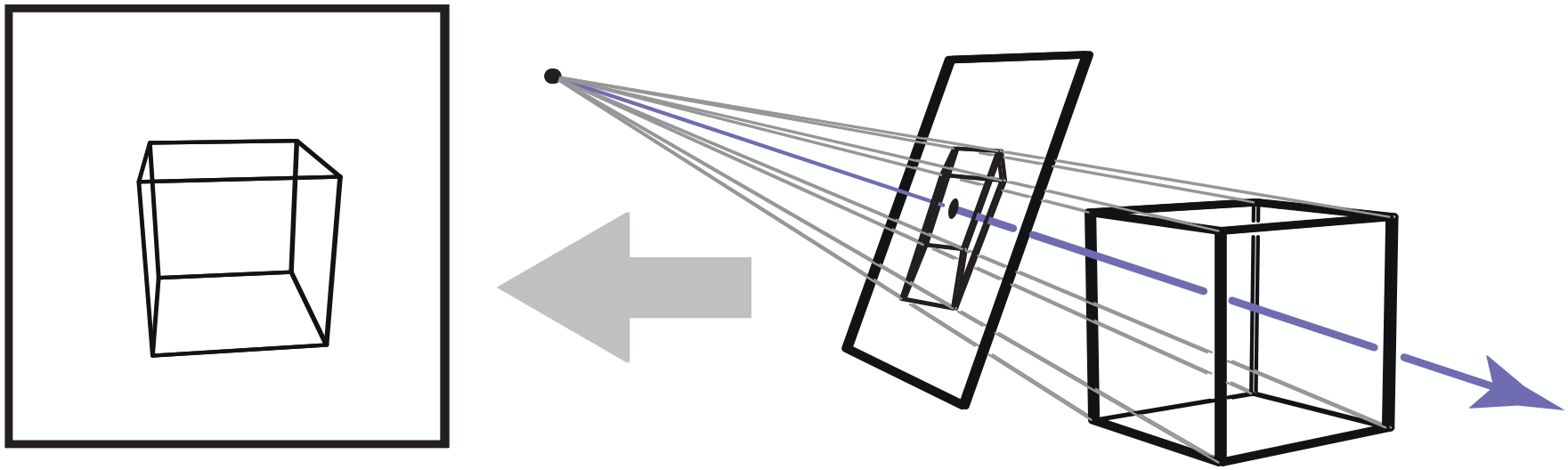


[source unknown]

© 2020 Fabio Pellacini

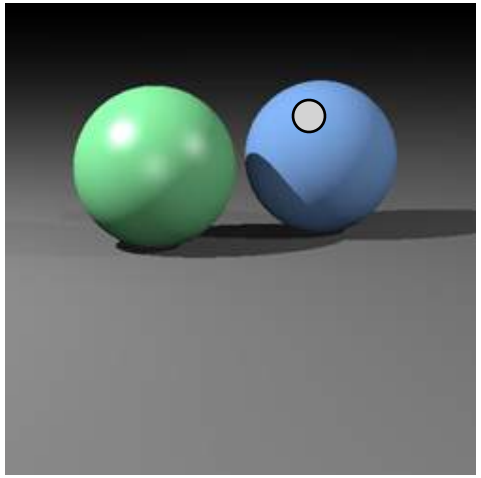
Ray tracing idea

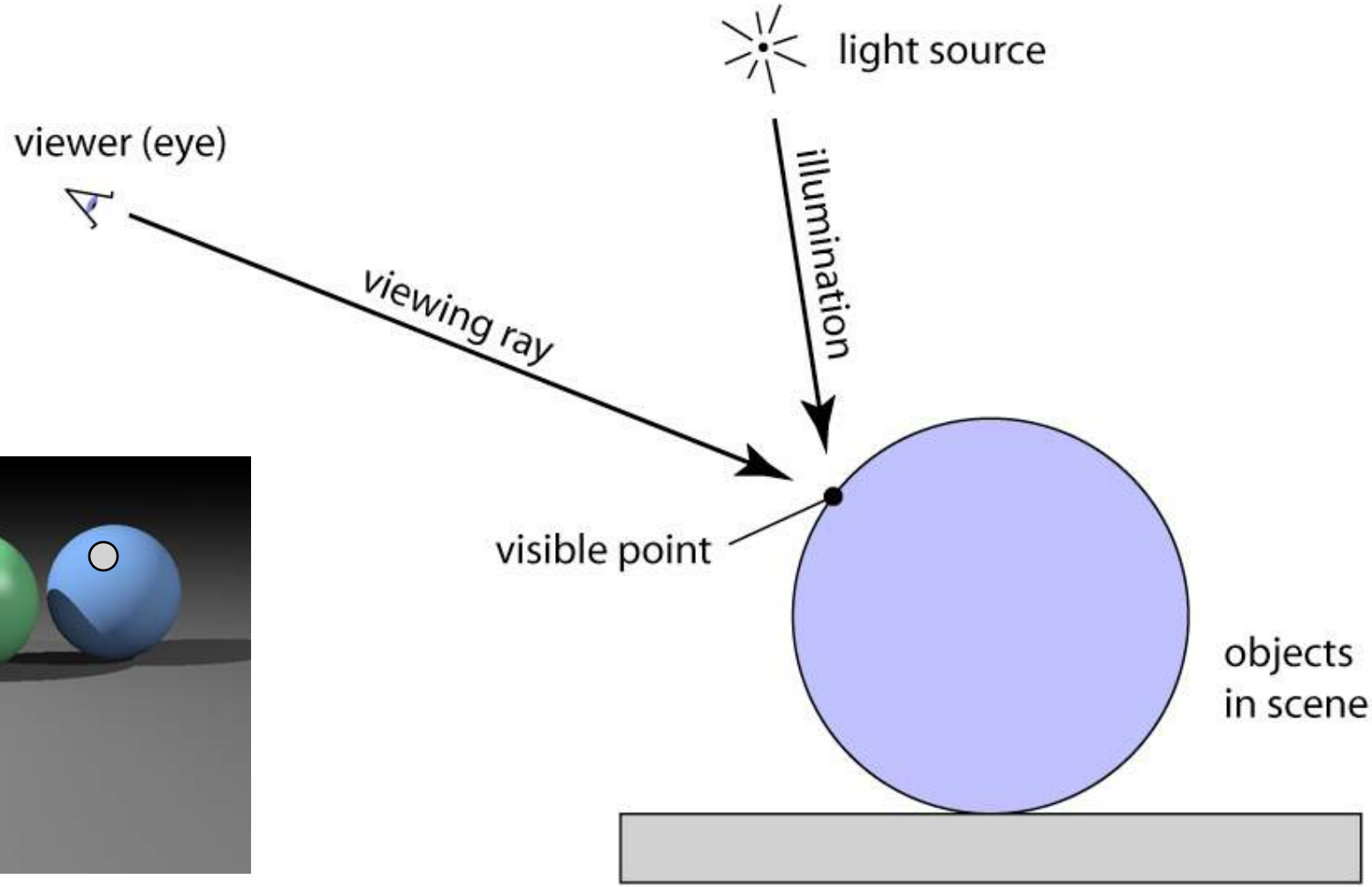
- Start with a pixel, search for surface visible in that pixel
- Set of points that project to a point in the image: a ray



© 2020 Fabio Pellacini

Ray tracing idea

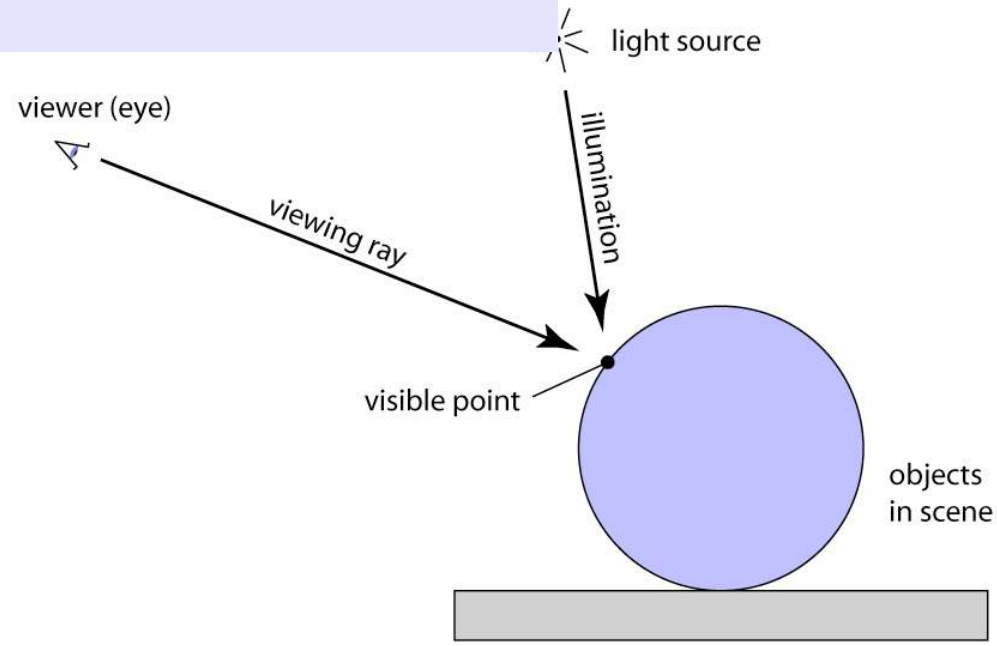




© 2020 Fabio Pellacini •

Ray tracing algorithm

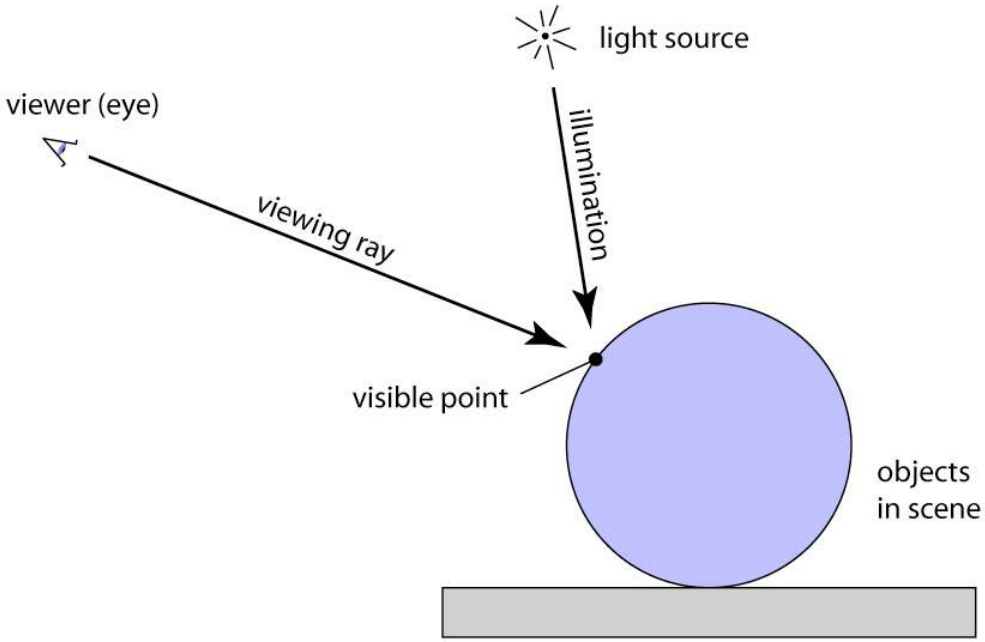
```
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    auto uv = (vec2f{i,j} + 0.5f) / img.size;
    auto ray = camera_ray(cam, uv);
    img[i,j] = shade(scene, ray);
  }
}
```



© 2020 Fabio Pellacini

Ray tracing algorithm

```
vec3f shade(scene* scene, ray3f ray) {
  auto point = intersect(scene, ray)
  Return compute_illumination(point)
}
```



© 2020 Fabio Pellacini •

Camera Rays

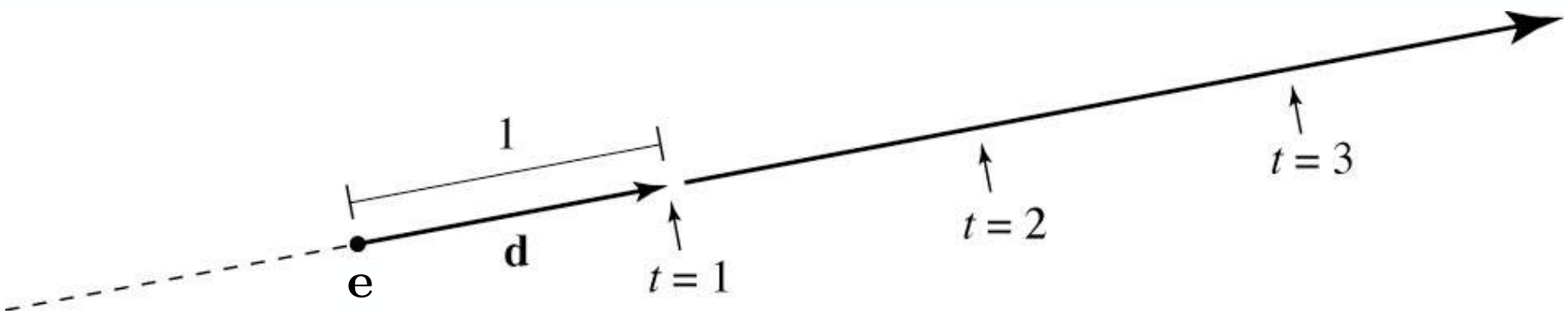
© 2020 Fabio Pellacini

Ray: a half line

- Standard representation: point **e** and direction **d**
  - this is a parametric equation for the line
  - add min and max values for *t*

```
struct ray3f {
  vec3f e, d;
  float tmin, tmax;
}
```

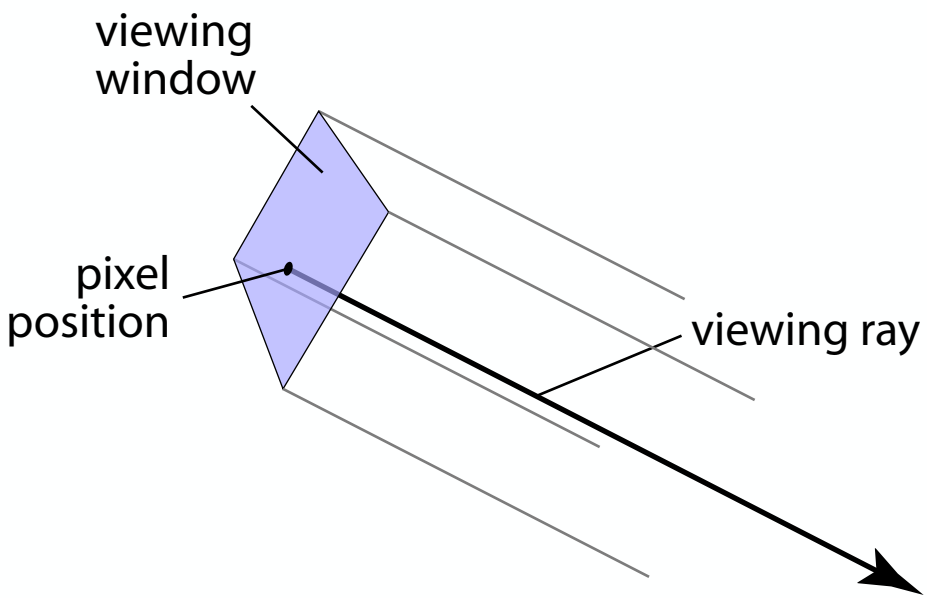
$$\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$$



© 2020 Fabio Pellacini

Orthographic projection

- Ray origin (varying): pixel position on viewing window
- Ray direction (constant): view direction

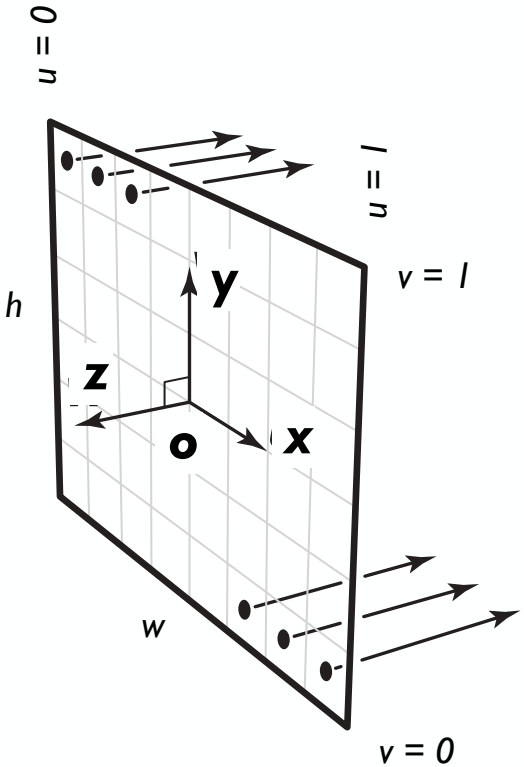


© 2020 Fabio Pellacini

Orthographic projection – rays

- Position and orient the camera with a frame $F = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}\}$
- Image plane of size *(w, h)* parametrized by *(u, v)* in [0,1]^2
- Define point on image plane **q** and with that the ray **r** as

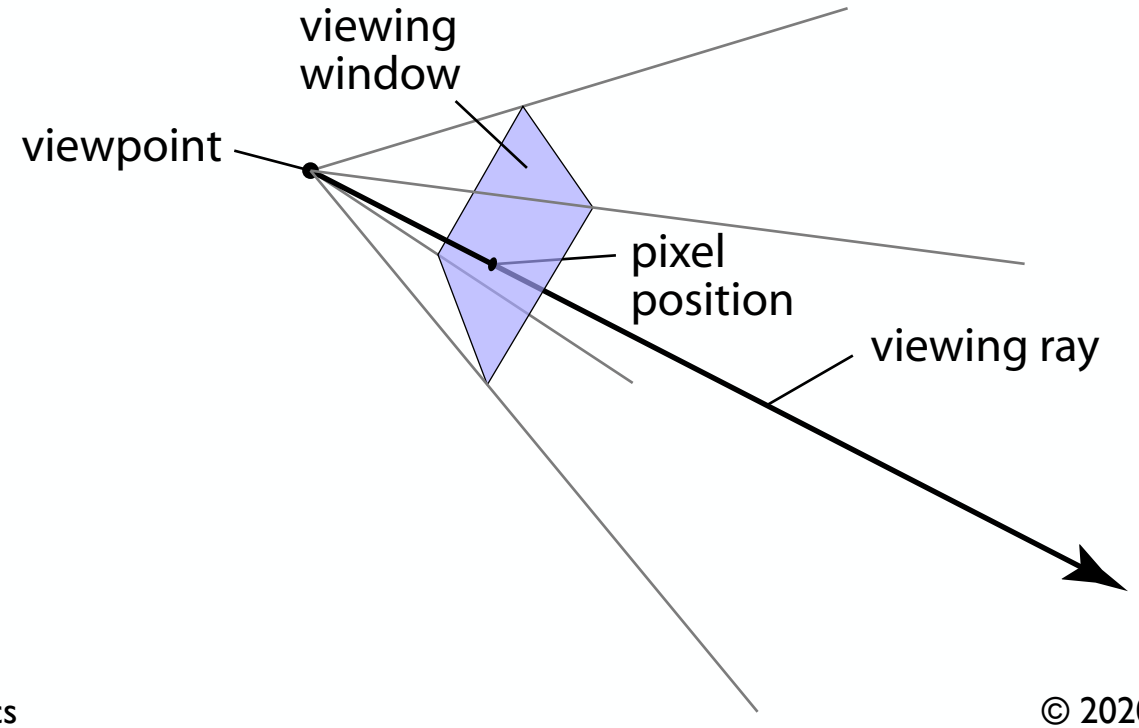
$$\mathbf{q} = \mathbf{o} + (u - 0.5)w\mathbf{x} + (v - 0.5)h\mathbf{y}$$
$$\mathrm{ray}(u, v) = \{\mathbf{q}, -\mathbf{z}\}$$
$$\text{with } (u, v) \in [0, 1]^2$$



© 2020 Fabio Pellacini

Perspective projection – rays

- Use window analogy directly
- Ray origin (constant): viewpoint
- Ray direction (varying): toward pixel position on viewing window



© 2020 Fabio Pellacini

Perspective projection – rays

- Position and orient the camera with a frame $F = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}\}$
- Image of size *(w, h) at* distance *d* parametrized by *(u, v)* in $[0,1]^2$
- Use field of view *fov* and aspect ratio *r* to define image size

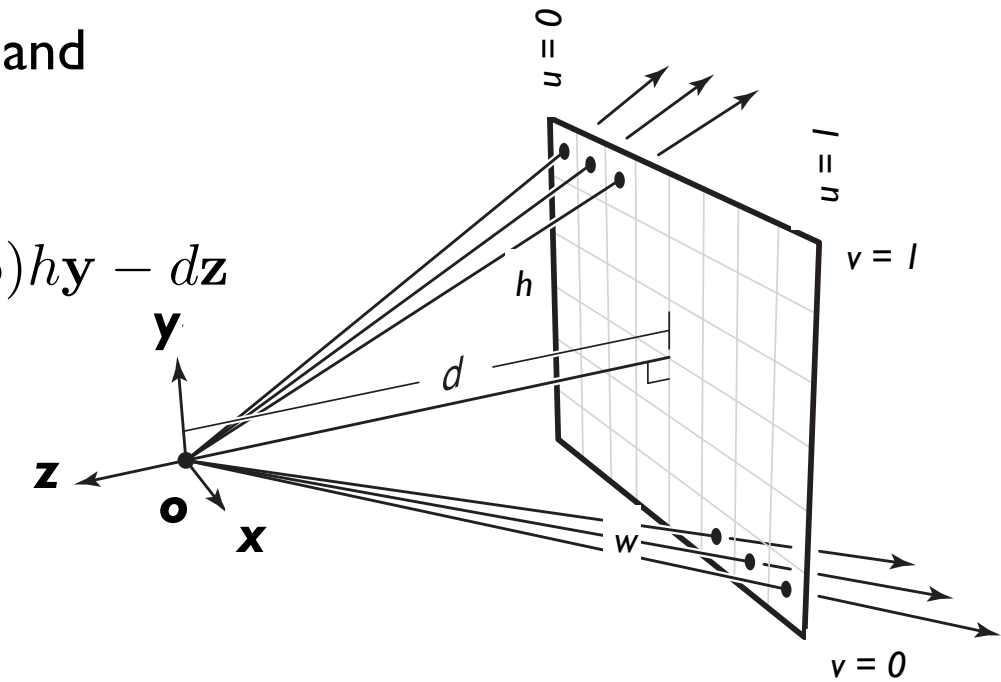
$$w = hr \quad h = 2d\tan(fov/2)$$

- Define point on image plane **q** and with that the ray **r** as

$$\mathbf{q} = \mathbf{o} + (u - 0.5)w\mathbf{x} + (v - 0.5)h\mathbf{y} - d\mathbf{z}$$

$$\mathrm{ray}(u, v) = \left\{\mathbf{o}, \frac{\mathbf{q} - \mathbf{o}}{|\mathbf{q} - \mathbf{o}|}\right\}$$

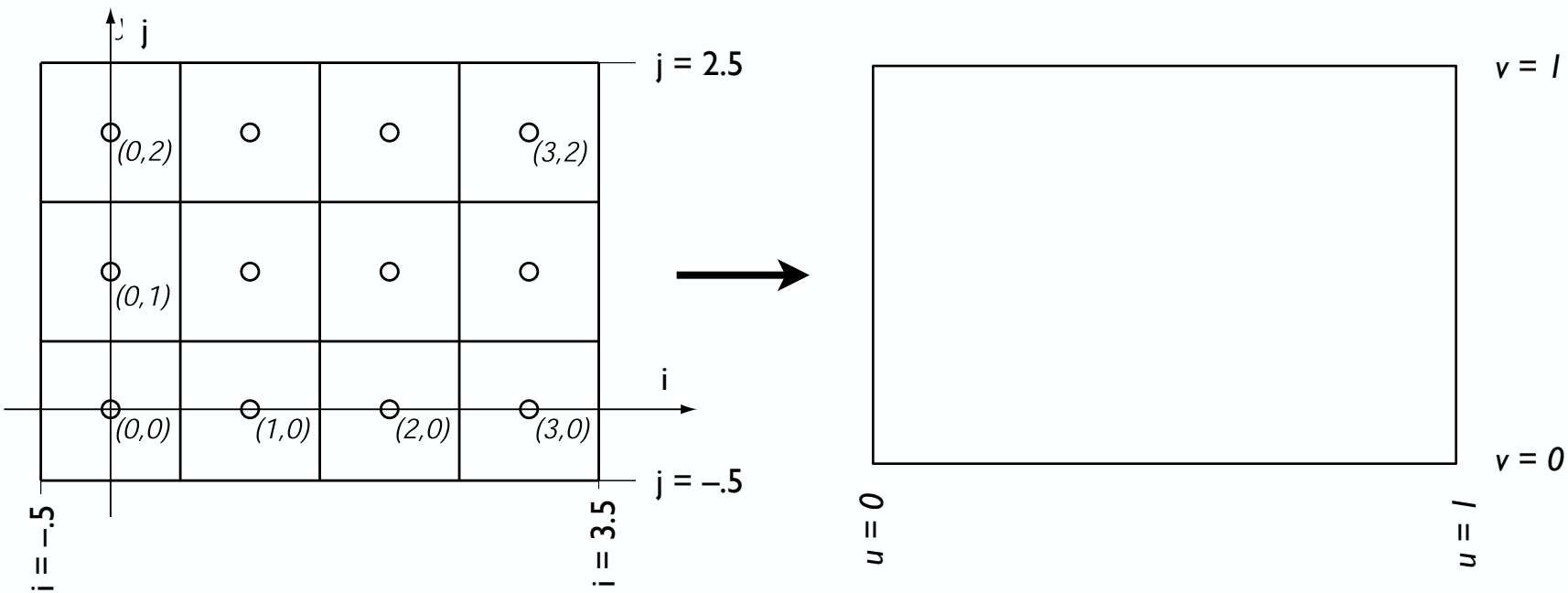
$$\text{with } (u, v) \in [0, 1]^2$$



© 2020 Fabio Pellacini

Pixel-to-image mapping

- Map pixels *(i,j)* to image plane coordinates *(u,v)*



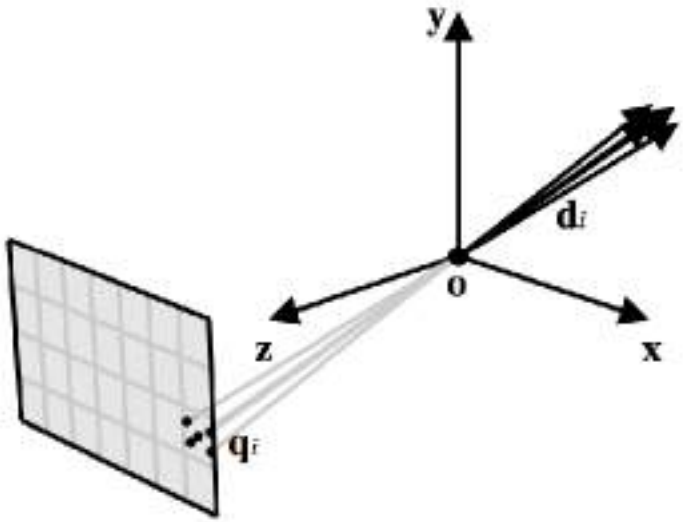
$$(u, v) = \left(\frac{i + 0.5}{n_x}, \frac{j + 0.5}{n_y} \right) \text{ with } (i, j) \in [0, n_x - 1] \times [0, n_y - 1]$$

© 2020 Fabio Pellacini

Pinhole camera

- Model an idealized pinhole camera with the sensor at the back. Same as the perspective model, but closer to real cameras.

```
ray3f eval_camera(camera* camera, vec2f image_uv) {
  auto q = vec3f{camera->film.x * (0.5 - image_uv.x),
    camera->film.y * (image_uv.y - 0.5), camera->lens};
  auto e = vec3f{0}; auto d = normalize(-q - e);
  return ray3f{transform_point(camera->frame, e),
      transform_direction(camera->frame, d)};
}
```

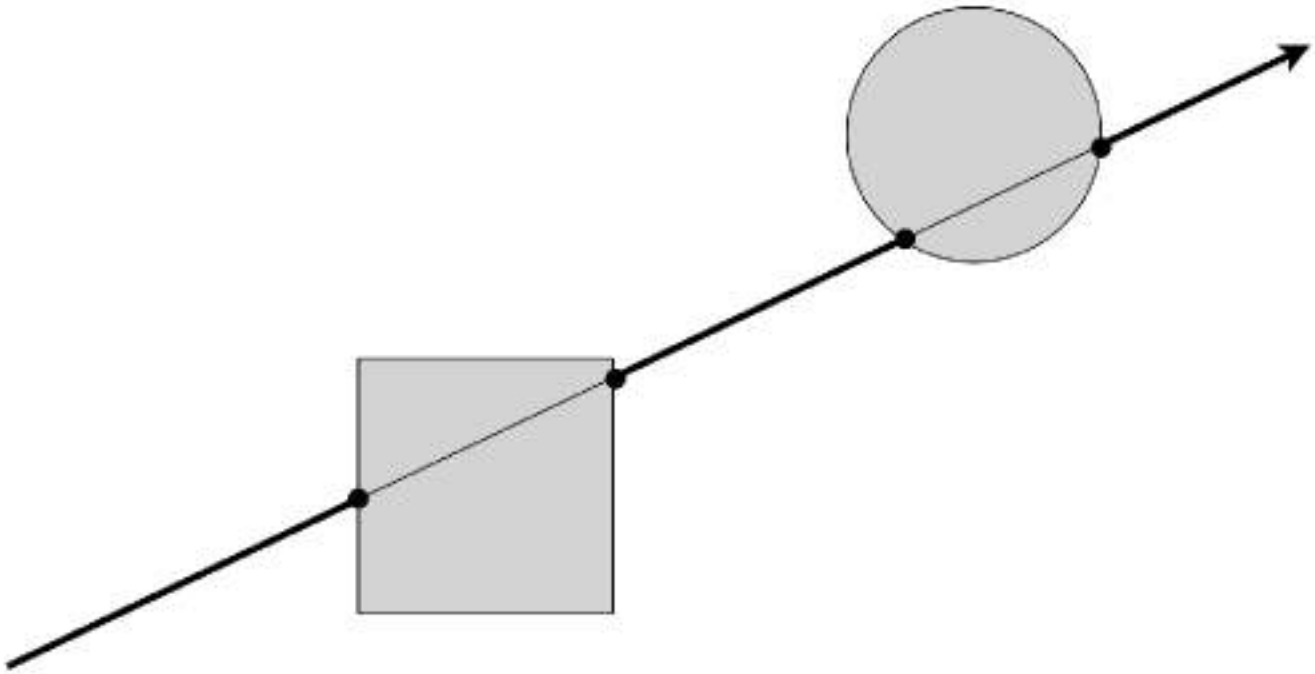


© 2020 Fabio Pellacini

Ray-Scene Intersection

© 2020 Fabio Pellacini

Ray intersection

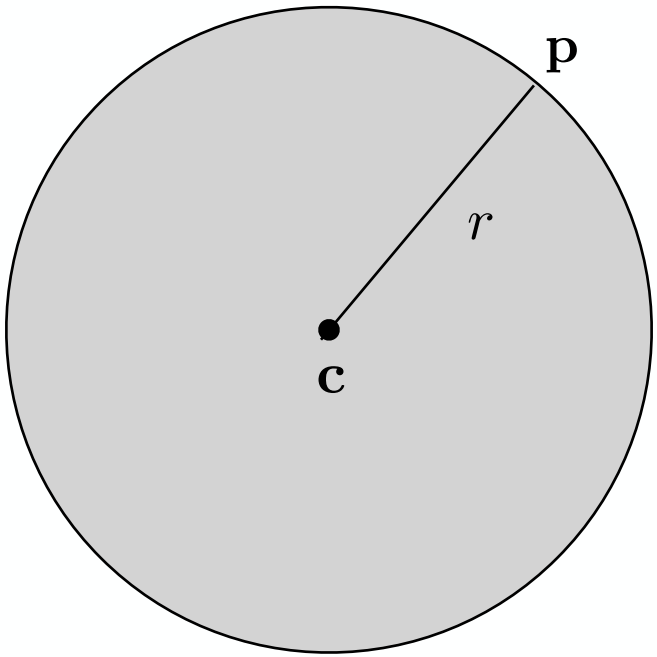


© 2020 Fabio Pellacini

Ray-sphere intersection

- Points on a sphere at at equal distance from its center

$$|\mathbf{p} - \mathbf{c}|^2 = r^2$$



© 2020 Fabio Pellacini

Ray-sphere intersection

- Enforce point on a ray and on a sphere
- Leads to system of equations

$$\begin{cases} \mathbf{p}(t) = \mathbf{e} + t\mathbf{d} \\ |\mathbf{p} - \mathbf{c}|^2 = r^2 \end{cases}$$

- By substitution

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) = r^2$$

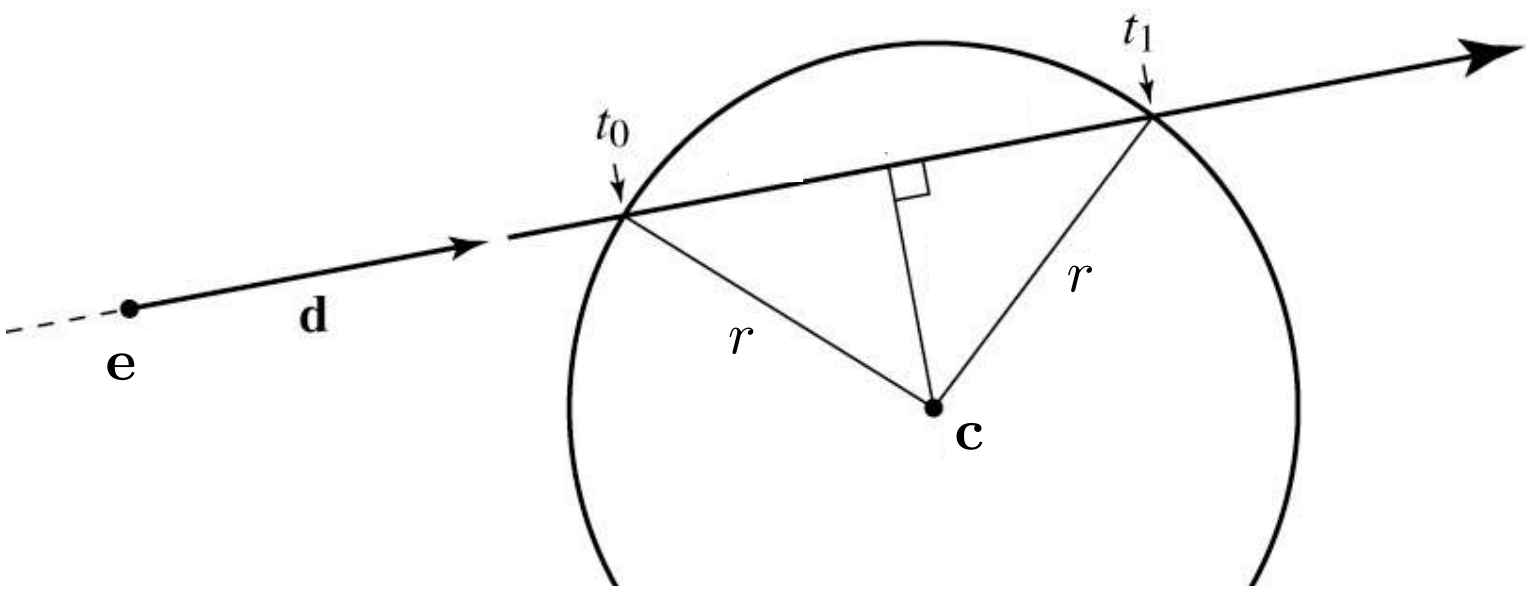
$$|\mathbf{d}|^2 t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + |\mathbf{e} - \mathbf{c}|^2 - r^2 = 0$$

© 2020 Fabio Pellacini

Ray-sphere intersection

- Algebraic equation $at^2 + bt + c = 0$
- With $a = |\mathbf{d}|^2 \quad b = 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}) \quad c = |\mathbf{e} - \mathbf{c}|^2 - r^2$
- No solution for $d = b^2 - 4ac < 0$
- Two solutions

$$t_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

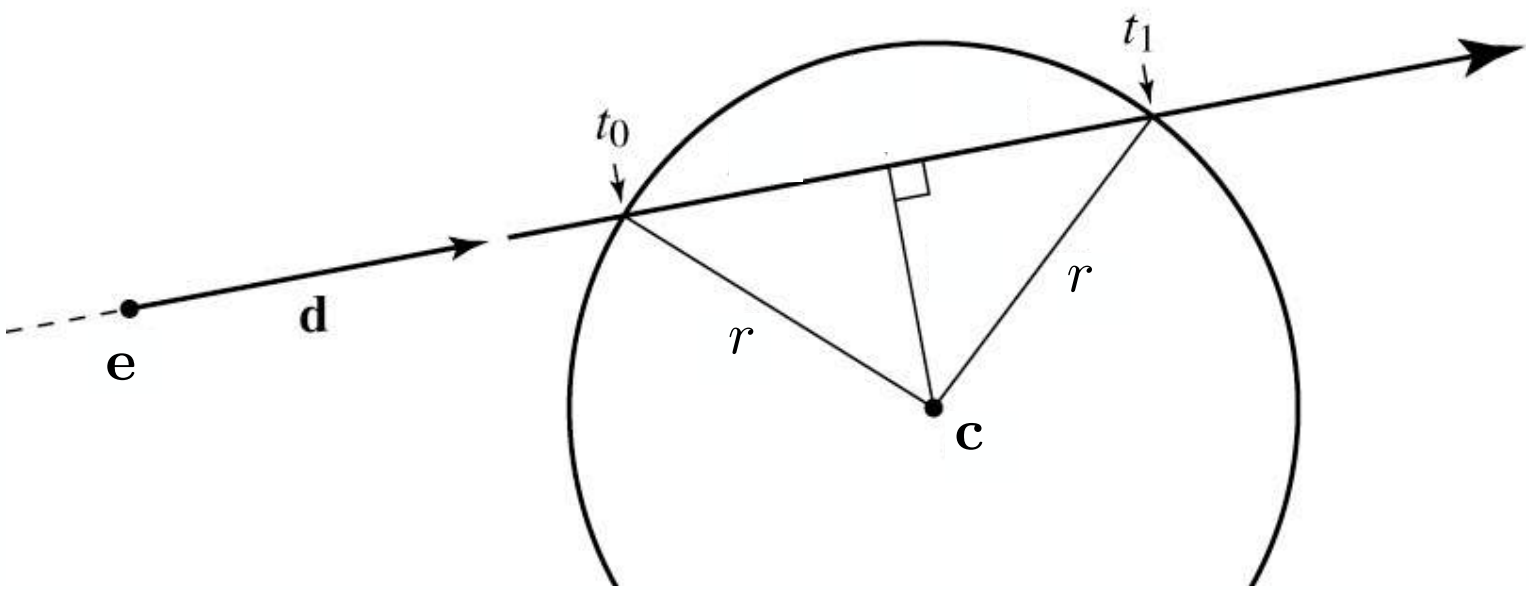


© 2020 Fabio Pellacini

Ray-sphere intersection

- Pick closest intersection within the ray bounds

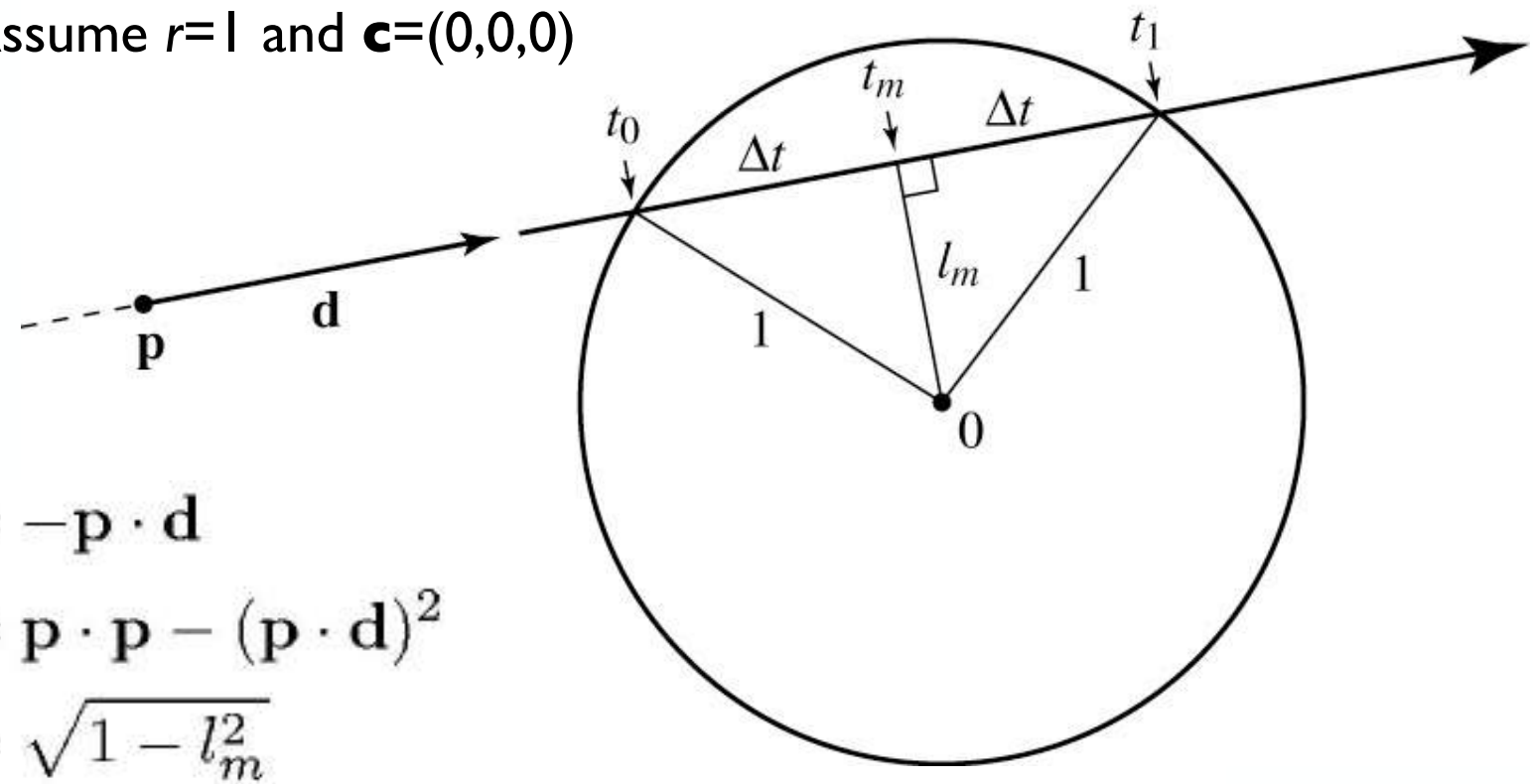
$$t_{min} \leq t_{\pm} \leq l_m ax$$



© 2020 Fabio Pellacini

Ray-sphere geometric

- Assume r=1 and $\mathbf{c}$=(0,0,0)



$$t_m = -\mathbf{p} \cdot \mathbf{d}$$

$$l_m^2 = \mathbf{p} \cdot \mathbf{p} - (\mathbf{p} \cdot \mathbf{d})^2$$

$$\Delta t = \sqrt{1 - l_m^2}$$

$$= \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

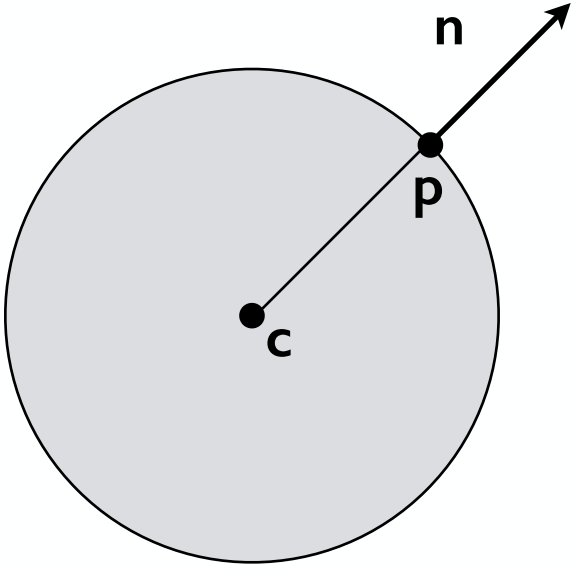
$$t_{0,1} = t_m \pm \Delta t = -\mathbf{p} \cdot \mathbf{d} \pm \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

© 2020 Fabio Pellacini

Sphere normals

- Shading depends on geometric normals
- For spheres, the normal is along the line that connects the sphere point to the sphere center

$$\mathbf{n} = \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|}$$



© 2020 Fabio Pellacini

Ray-triangle intersection

- Use algebraic view since it is significantly simpler
- Write point w.r.t. barycentric coordinates and save for those
- Also used later for texturing

© 2020 Fabio Pellacini

Ray-triangle intersection

- Every point on the triangle can be as

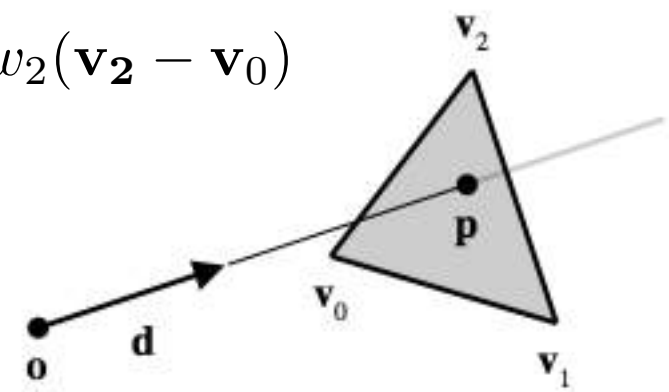
$$\mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

- Set up a system of equations

$$\begin{cases} \mathbf{p}(t) = \mathbf{e} + \mathbf{d}t \\ \mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v_2} - \mathbf{v}_0) \end{cases}$$

- By substitution

$$\mathbf{e} + \mathbf{d}t = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v_2} - \mathbf{v}_0)$$



© 2020 Fabio Pellacini

Ray-triangle intersection

- Linear system of 3 equations for 3 unknowns

$$\mathbf{e} + \mathbf{d}t = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

$$w_1(\mathbf{v}_0 - \mathbf{v}_1) + w_2(\mathbf{v}_0 - \mathbf{v}_2) + \mathbf{d}t = \mathbf{v}_0 - \mathbf{e}$$

$$\begin{bmatrix} \mathbf{v}_0 - \mathbf{v}_1 & \mathbf{v}_0 - \mathbf{v}_2 & \mathbf{d} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 - \mathbf{e} \end{bmatrix}$$

- Solve with Cramer's rule

$$t = \frac{(\mathbf{v}_2 \times \mathbf{e}_1) \cdot \mathbf{e}_2}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \quad w_1 = \frac{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{v}_2}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \quad w_2 = \frac{(\mathbf{v}_2 \times \mathbf{e}_1) \cdot \mathbf{d}}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1}$$

$$\text{with} \quad \mathbf{v}_2 = \mathbf{e} - \mathbf{v}_0 \quad \mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0 \quad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$$

- Check for constraints on barycentric coordinates and ray distance

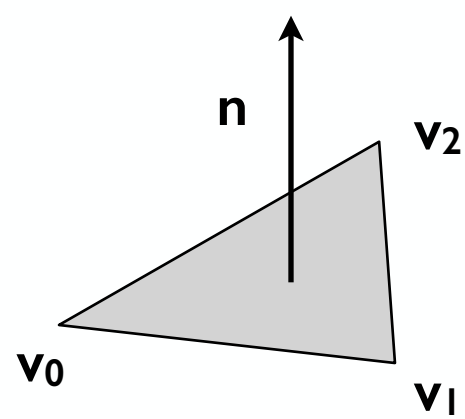
$$t_{min} \leq t \leq t_{max} \quad 0 \leq w_1 \leq 1 \quad 0 \leq w_2 \leq 1 \quad w_1 + w_2 \leq 1$$

© 2020 Fabio Pellacini

Triangle normal

- Shading depends on geometric normals
- For triangles, the normal is the normal of the plane where the triangle lies
- Normals can computed from the triangle vertices as

$$\mathbf{n} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}$$



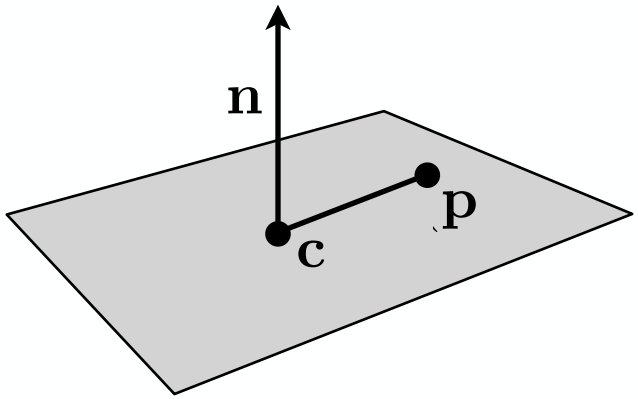
© 2020 Fabio Pellacini

Infinite plane representation

- Vector between two points on the plane is orthogonal to the plane normal

$$(\mathbf{p} - \mathbf{c}) \cdot \mathbf{n} = 0$$

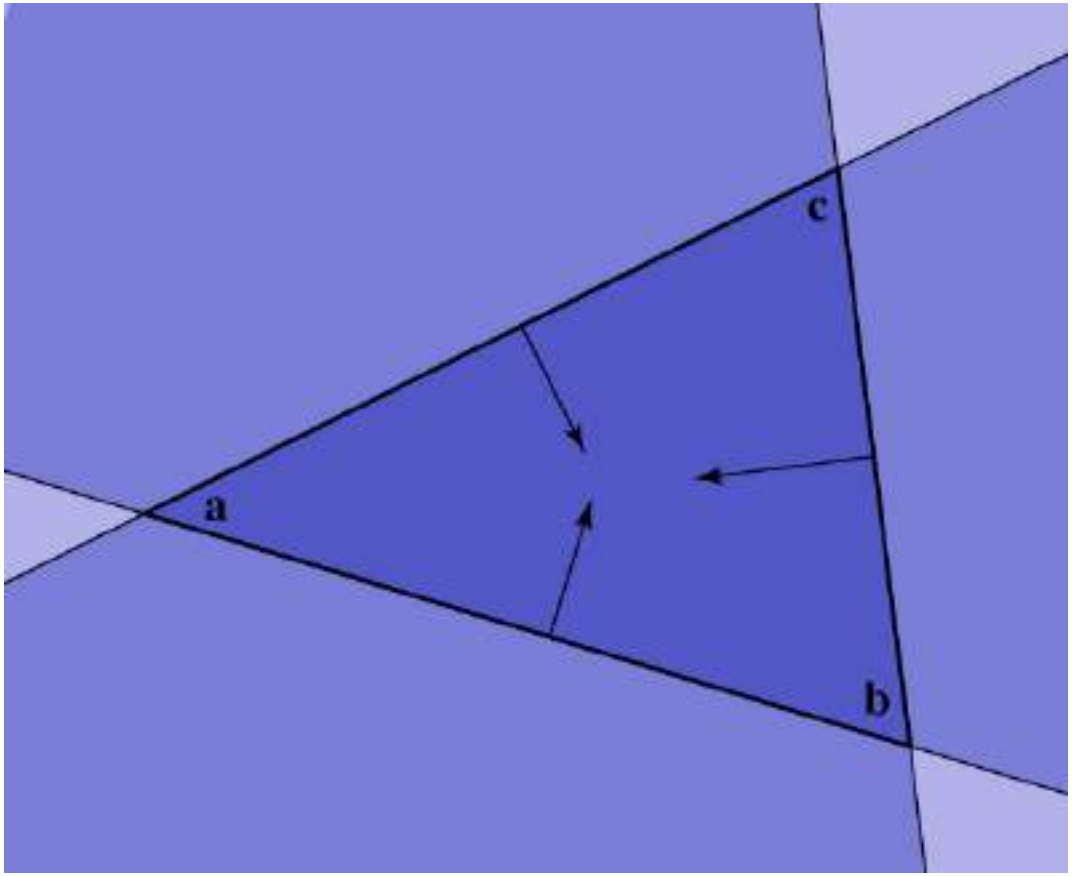
$$\mathbf{p} \cdot \mathbf{n} + k = 0$$



© 2020 Fabio Pellacini

Ray-triangle geometric

- First compute the intersection with the infinite plane
- In plane, triangle is the intersection of 3 half spaces



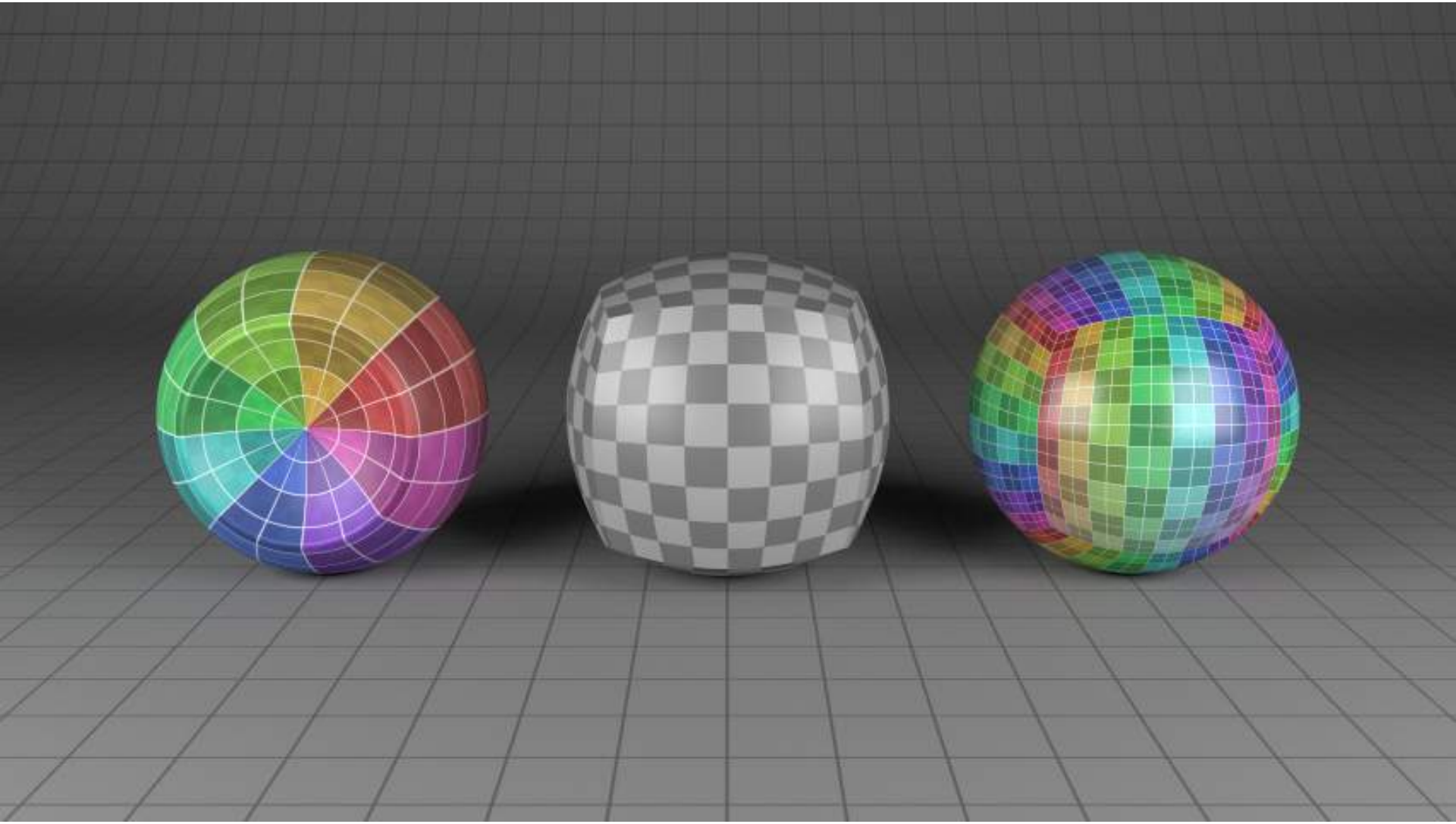
© 2020 Fabio Pellacini

Ray-triangle intersection

```
bool intersect_triangle(ray3f ray, vec3f p0, vec3f p1,
  vec3f p2, vec2f* uv, float* dist) {
  // compute triangle edges
  auto edge1 = p1 - p0; auto edge2 = p2 - p0;
  // compute determinant to solve a linear system
  auto pvec = cross(ray.d, edge2); auto det = dot(edge1, pvec);
  // check determinant and exit if triangle and ray are parallel
  if (det == 0) return false; // can use epsilon
  auto idet = 1.0f / det;
  // compute and check first barycentric coordinate
  auto tvec = ray.o - p0;  auto u = dot(tvec, pvec) * idet;
  if (u < 0 || u > 1) return false;
  // compute and check second barycentric coordinate
  auto qvec=cross(tvec,edge1); auto v = dot(ray.d, qvec) * idet;
  if (v < 0 || u + v > 1) return false;
  // compute and check ray parameter
  auto t = dot(edge2, qvec) * idet;
  if (t < ray.tmin || t > ray.tmax) return false;
  // intersection occurred: set params and exit
  *uv = {u, v}; *dist = t; return true;
}
```


© 2020 Fabio Pellacini

Ray-triangle intersection



© 2020 Fabio Pellacini

Ray-point intersection

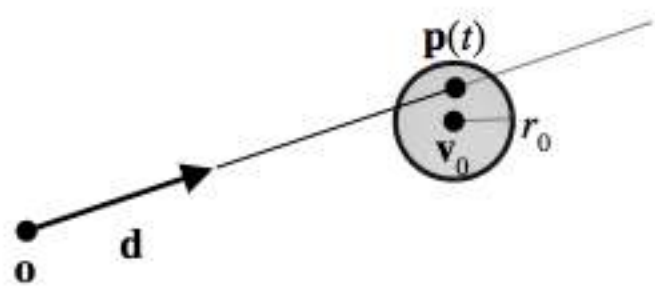
- *Approximate* solution since points are really unidimensional
- Find ray-point distance and then check whether it is less then segment radius

$$\frac{d}{dt}D^2(t) = \frac{d}{dt}|(\mathbf{e} + t\mathbf{d}) - \mathbf{v}_0|^2 = 0 \rightarrow$$

$$2\mathbf{d} \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{v}_0) = 0 \rightarrow$$

$$t = \frac{(\mathbf{v}_0 - \mathbf{o}) \cdot \mathbf{d}}{\mathbf{d} \cdot \mathbf{d}}$$

- Then clamp *t* in the valid range and check with point radius



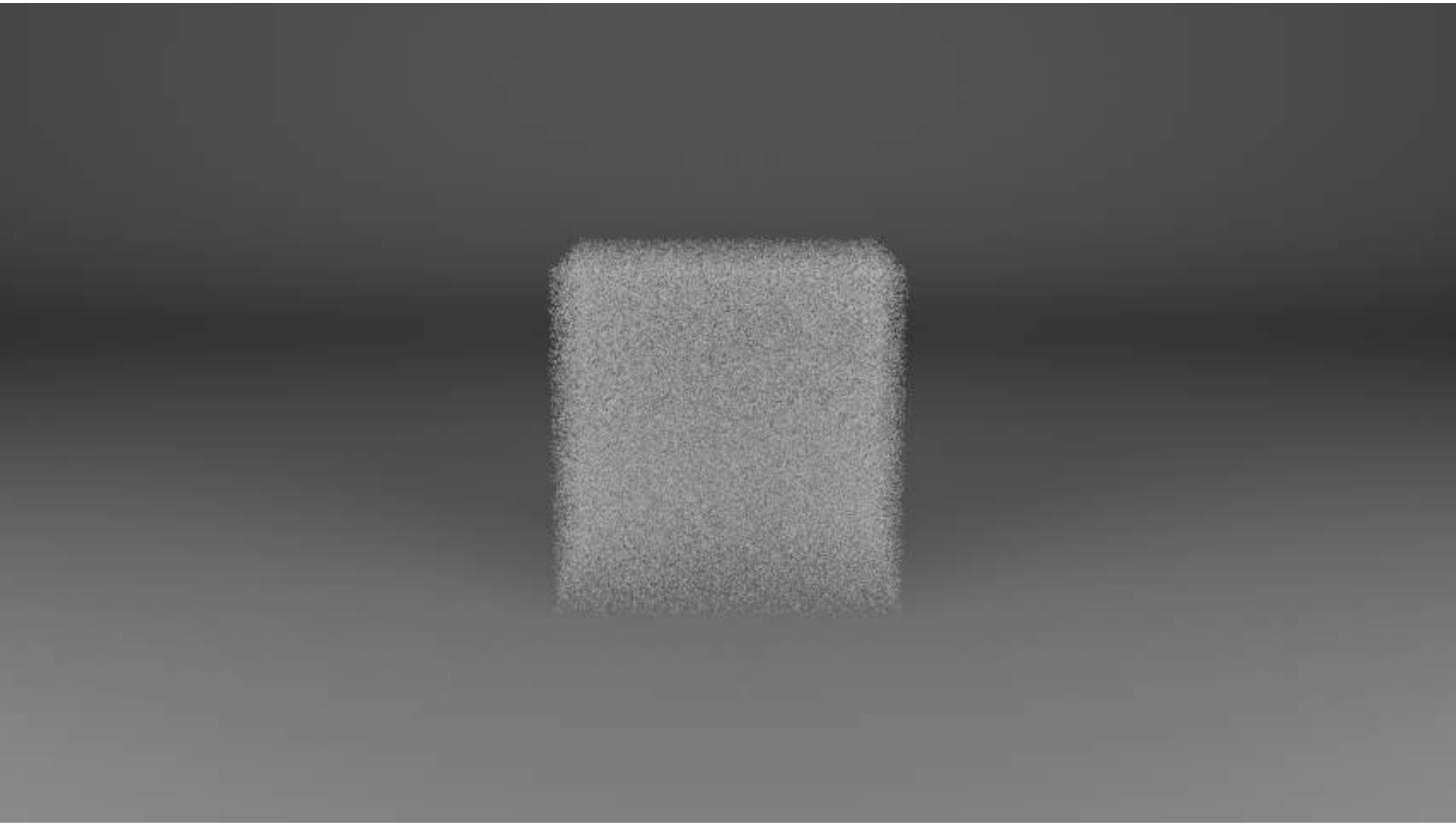
© 2020 Fabio Pellacini

Ray-point intersection

```
bool intersect_point(ray3f ray, vec3f p, float r,
  vec2f* uv, float* dist) {
  // find parameter for line-point minimum distance
  auto w = p - ray.o;
  auto t = dot(w, ray.d) / dot(ray.d, ray.d);
  // exit if not within bounds
  if (t < ray.tmin || t > ray.tmax) return false;
  // test for line-point distance vs point radius
  auto rp  = ray.o + ray.d * t;
  auto prp = p - rp;
  if (dot(prp, prp) > r * r) return false;
  // intersection occurred: set params and exit
  *uv = {0, 0}; *dist = t; return true;
}
```


© 2020 Fabio Pellacini

Ray-point intersection

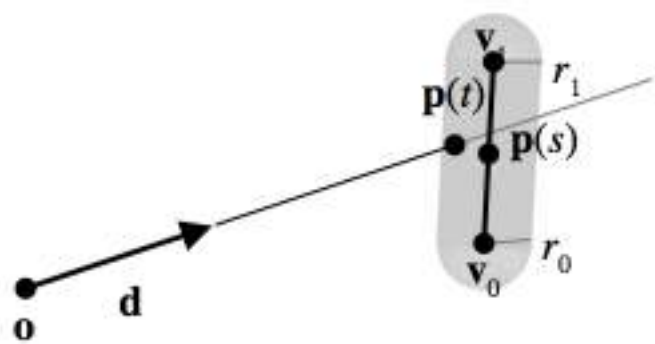


© 2020 Fabio Pellacini

Ray-segment intersection

- *Approximate* solution since lines are really unidimensional
- Find the closest points on ray and segment and then check whether their distance is less then segment radius
- Compute the points by minimizing the distance over the ray and segment parameters

$$D^2(s,t) = |(\mathbf{e} + t\mathbf{d}) - (\mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0))|^2 \rightarrow (s,t) = \arg\min D^2(s,t)$$



© 2020 Fabio Pellacini

Ray-line intersection

- Solve by setting derivatives to zero

$$\begin{cases} \partial D^2(s,t)/\partial s = 0 \\ \partial D^2(s,t)/\partial t = 0 \end{cases} \rightarrow$$

$$\begin{cases} 2(\mathbf{v}_1 - \mathbf{v}_0) \cdot [(\mathbf{e} + t\mathbf{d}) - (\mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0))] = 0 \\ 2\mathbf{d} \cdot [(\mathbf{e} + t\mathbf{d}) - (\mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0))] = 0 \end{cases} \rightarrow$$

$$s = \frac{be - cd}{ac - b^2} \quad t = \frac{ae - bd}{ac - b^2} \quad \text{with}$$

$$a = \mathbf{d} \cdot \mathbf{d} \quad b = \mathbf{d} \cdot (\mathbf{v}_1 - \mathbf{v}_0) \quad c = (\mathbf{v}_1 - \mathbf{v}_0) \cdot (\mathbf{v}_1 - \mathbf{v}_0)$$

$$d = \mathbf{d} \cdot (\mathbf{e} - \mathbf{v}_0) \quad e = (\mathbf{v}_1 - \mathbf{v}_0) \cdot (\mathbf{e} - \mathbf{v}_0)$$

- We then clamp *s and t* in the appropriate ranges and check whether their distance is less then the segment radius

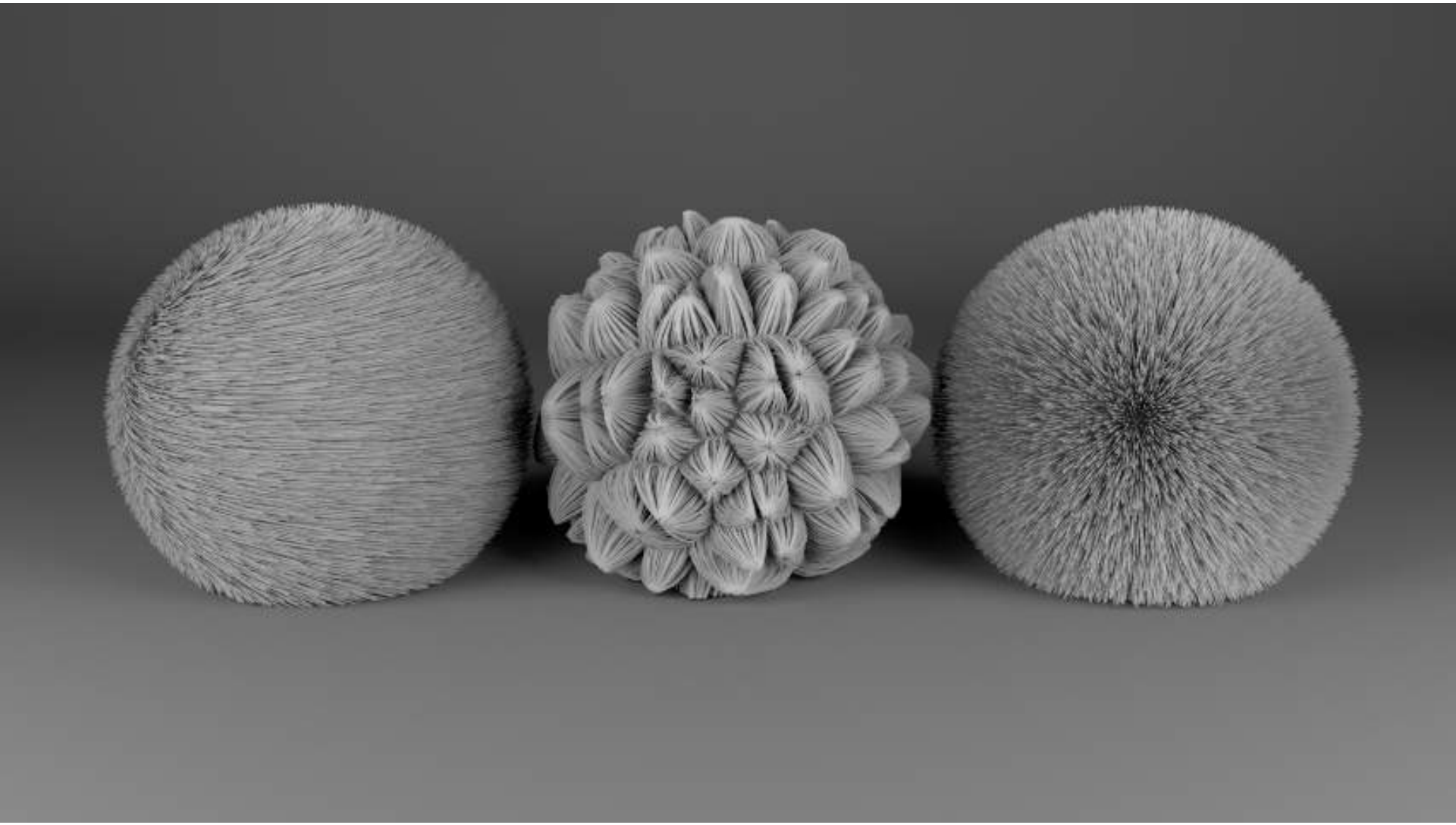
© 2020 Fabio Pellacini

Ray-line intersection

```
bool intersect_line(ray3f ray, vec3f p0, vec3f p1, float r0,
  float r1, vec2f* uv, float* dist) {
  // setup intersection params
  auto u = ray.d; auto v = p1 - p0; auto w = ray.o - p0;
  // compute values to solve a linear system
  auto a = dot(u, u); auto b = dot(u, v); auto c   = dot(v, v);
  auto d = dot(u, w); auto e = dot(v, w); auto det = a*c - b*b;
  // check determinant and exit if lines are parallel
  if (det == 0) return false; // can use epsilon
  // compute distance on ray and segment, clamped to corners
  auto t = (b*e-c*d)/det; auto s = clamp((a*e-b*d)/det, 0, 1);
  // exit if not within bounds
  if (t < ray.tmin || t > ray.tmax) return false;
  // compute segment-segment distance on the closest points
  auto pr=ray.o+ray.d*t; auto pl=p0+(p1-p0)*s; auto prl=pr-pl;
  // check with the line radius at the same point
  auto d2 = dot(prl, prl); auto r  = r0 * (1 - s) + r1 * s;
  if (d2 > r * r) return false;
  // intersection occurred: set params and exit
  *uv = {s, sqrt(d2) / r}; *dist = t; return true;
}
```


© 2020 Fabio Pellacini

Ray-line intersection



© 2020 Fabio Pellacini

Interpolation in ray tracing

- When values are stored at vertices, use barycentric interpolation to define values across the whole surface that:
  - match the values at the vertices
  - are continuous across edges
  - are piecewise linear (linear over each triangle)
  - we assume that vertex values do not differ much
- How to compute interpolated values
  - during triangle intersection compute barycentric coords
  - use barycentric coords to average attributes given at vertices

© 2020 Fabio Pellacini

Ray intersection in software

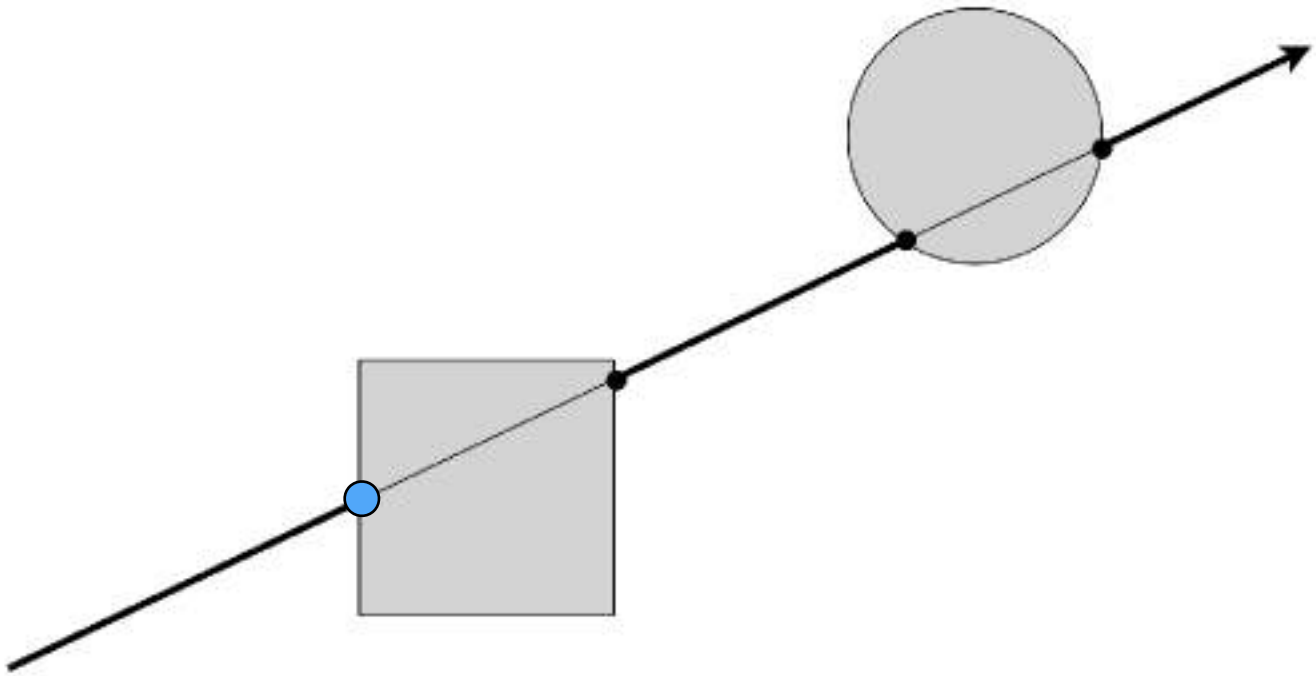
- Option 1: All shapes intersectable via virtual functions
  - but virtual functions are slow
- Option 2: Focus on some shapes only, use external functions
  - much faster and common in real systems, e.g. Intel Embree
  - basic shapes: triangles, lines, points
  - extended shapes: curves, subdivision surfaces
  - returns whether we intersect, the ray distance and element barycentric coordinates

```
bool intersect_triangle(ray3f r, vec3f a, vec3f b,
  vec3f c, float* ray_t, vec2f* uv);
```


© 2020 Fabio Pellacini

Ray-scene intersection

- Scenes have many instances: search for the *closest* intersection
- Loop over instances, ignoring those that don't intersect



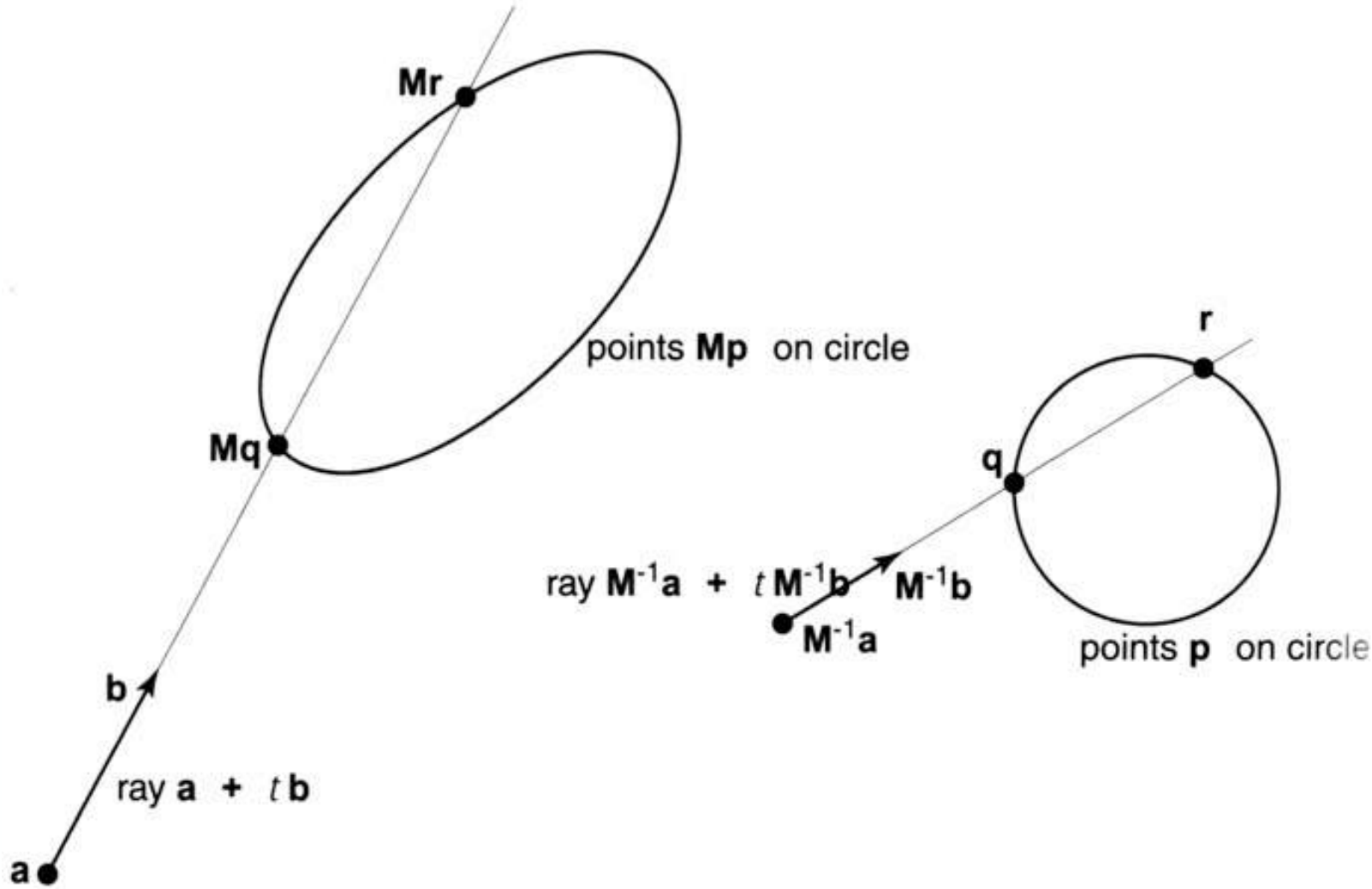
© 2020 Fabio Pellacini

Ray-instance intersection

- Each instance define its own coordinate system
  - mesh vertices are written w.r.t. a local coordinate system
- Rays are in world coordinates
- So we cannot intersect ray with triangle instances directly
- Option 1: transform each triangle to world coordinates
  - too expensive since we have many triangles
- Option 2: transform the ray in local coordinates
  - ray distance remains the same
  - ray origin and direction are transformed

© 2020 Fabio Pellacini

Ray-instance intersection



© 2020 Fabio Pellacini

Ray-scene intersection

- Return a record of the intersected instance, its shape element and the shape barycentric coordinates

```
struct intersection3f {
    float distance; // ray distance
    int object;     // object id
    int element;    // element id (triangle, line, point)
    vec2f uv;       // barycentric coordinates
};
```


© 2020 Fabio Pellacini

Ray-scene intersection

- Convenient implementation: keep track of the closest shape by updating the *tmax* value with the closest one
  - this way we skip all further shapes with the shape checks

```
bool intersect(scene* scene, ray3f ray,
    intersection3f* isec) {
  auto hit = false;
  for(auto object_id : range(scene->objects.size())) {
    auto object = scene->objects[object_id];
    auto tray = transform_ray(inverse(object->frame), ray);
    if(!intersect(object->shape, tray, isec)) continue;
    hit = true;
    isec->object = object_id;
    ray.tmax = isec->ray_t;
  }
  return hit;
}
```


© 2020 Fabio Pellacini

Ray-shape intersection

- Same as scene, but no need to transform ray
- Example here for triangle mesh

```
bool intersect(shape* shape, ray3f ray,
    intersection3f* isec) {
  auto hit = false;
  for(auto element_id : range(shape->triangles.size())) {
    auto t = shape->triangles[element_id];
    if(!intersect_triangle(ray, shape->pos[t.x],
        shape->pos[t.y], shape->pos[t.z],
        isec->distance, isec->uv)) continue;
    hit = true;
    isec->element = element_id;
    ray.tmax = isec->distance;
  }
  return hit;
}
```


© 2020 Fabio Pellacini

Ray-scene intersection

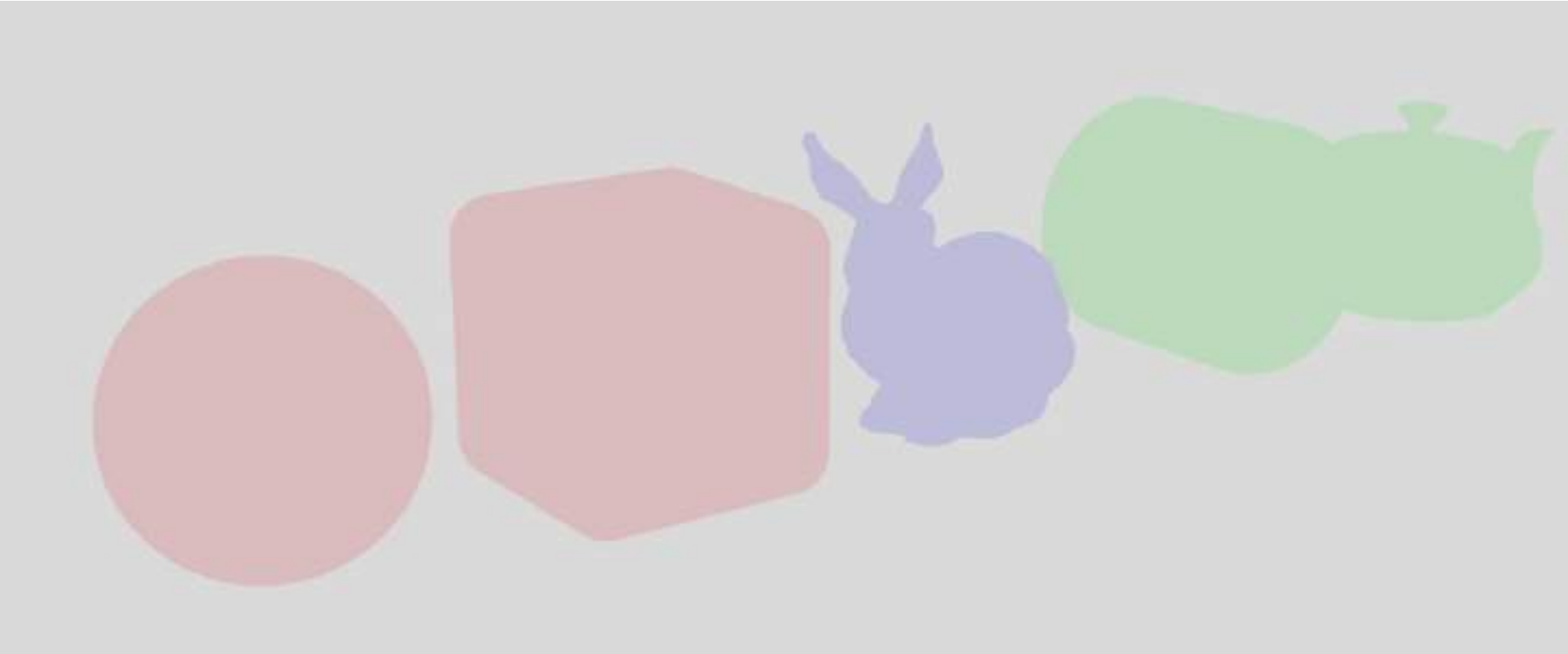
- Image with eye ray generation and scene intersection

```
for(auto j : range(height)) {
  for(auto i : range(width)) {
    auto (u, v) = pixel_uv(i,j);
    auto ray = camera_ray(u,v);
    img[i,j] = shade(scene, ray);
  }
}

vec3f shade_color(scene* scene, ray3f ray) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return zero3f;
  auto object = scn->objects[isec.object];
  return object->material->color;
}
```


© 2020 Fabio Pellacini

Ray-scene intersection

- Image with eye ray generation and scene intersection



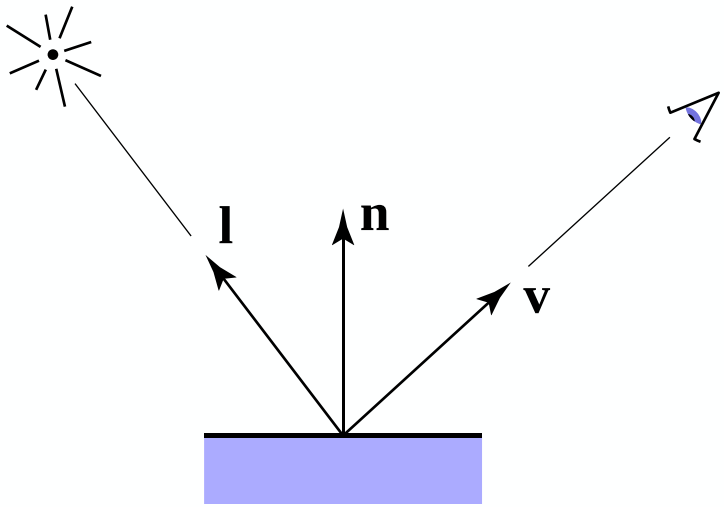
© 2020 Fabio Pellacini

Shading

© 2020 Fabio Pellacini

Shading

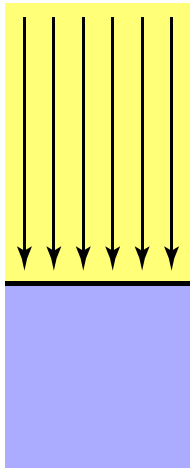
- Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction
    (for each of many lights)
  - surface normal
  - surface parameters
    (color, shininess, …)



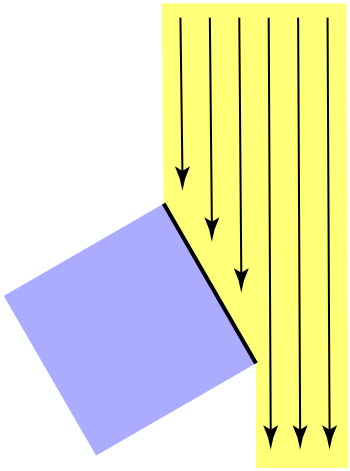
© 2020 Fabio Pellacini

Diffuse reflection

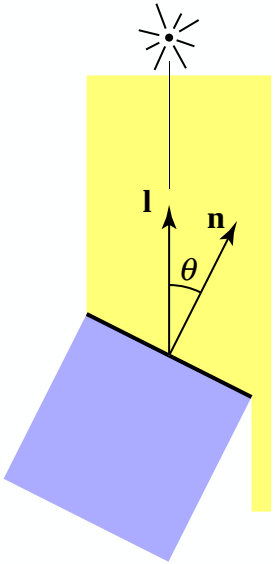
- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law



Top face of cube receives a certain amount of light



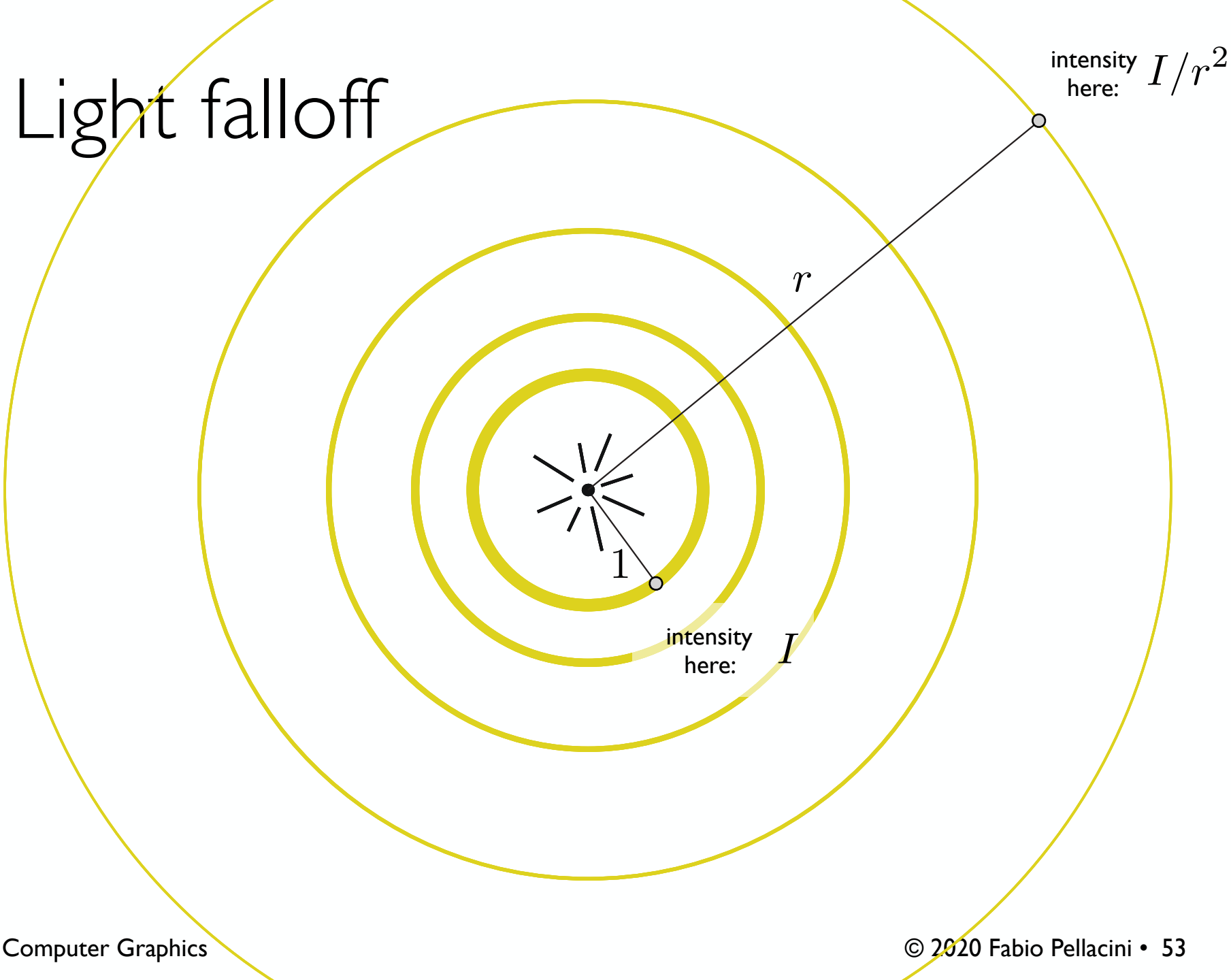
Top face of 60° rotated cube intercepts half the light



In general, light per unit area is proportional to $\cos \theta = l \cdot n$

© 2020 Fabio Pellacini

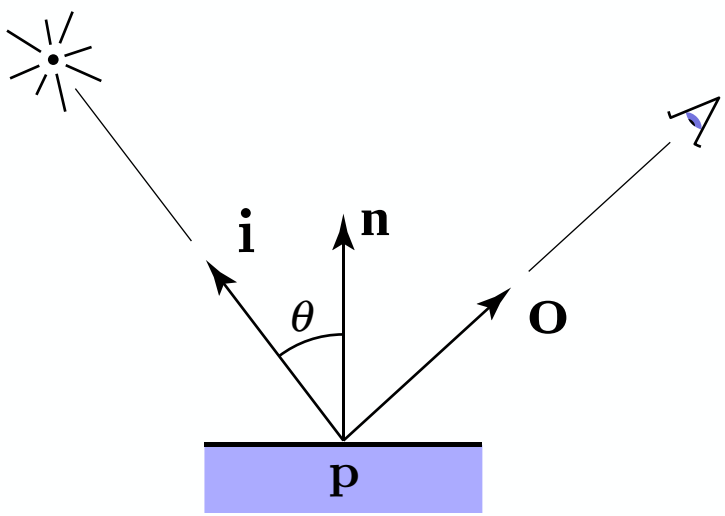
Light falloff



© 2020 Fabio Pellacini

Diffuse illumination

- Shading depends on lighting direction and normal
- Normalize by pi as discussed later



illumination from source

$$L_d = \frac{k_d}{\pi} \frac{I}{|\mathbf{s}-\mathbf{p}|^2} max(0, \mathbf{n} \cdot \mathbf{i})$$

diffuse coefficient

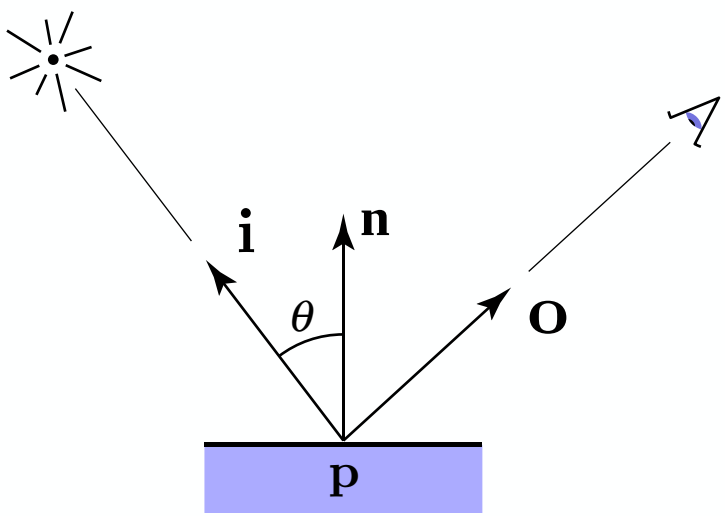
diffusely reflected illumination

$$\mathbf{i} = \frac{\mathbf{s}-\mathbf{p}}{|\mathbf{s}-\mathbf{p}|}$$

© 2020 Fabio Pellacini

Diffuse illumination

- Take the absolute value of the dot product to ignore sidedness
- Sum contribution of multiple lights



sum over lights

$$L_d = \sum_i \frac{k_d}{\pi} \frac{I_i}{|\mathbf{s}_i - \mathbf{p}|^2} |\mathbf{n} \cdot \mathbf{i}_i|$$

ignore sideness

© 2020 Fabio Pellacini

Computing shape values

- First, evaluate normal and position from intersection

```
T interpolate_triangle(T a, T b, T c, vec2f uv) {
  return a * (1 - uv.x - uv.y) + b * uv.x + c * uv.y;
}

vec3f eval_normal(shape* shape, int element, vec2f uv) {
  auto t = triangles[element];
  return normalize(interpolate_triangle(
    shape->normals[t.x], shape->normals[t.y],
    shape->normals[t.z], uv));
}

vec3f eval_position(shape* shape, int element, vec2f uv) {
  auto t = triangles[element];
  return interpolate_triangle(shape->positions[t.x],
    shape->positions[t.y], shape->positions[t.z], uv);
}
```


© 2020 Fabio Pellacini

Normal shading

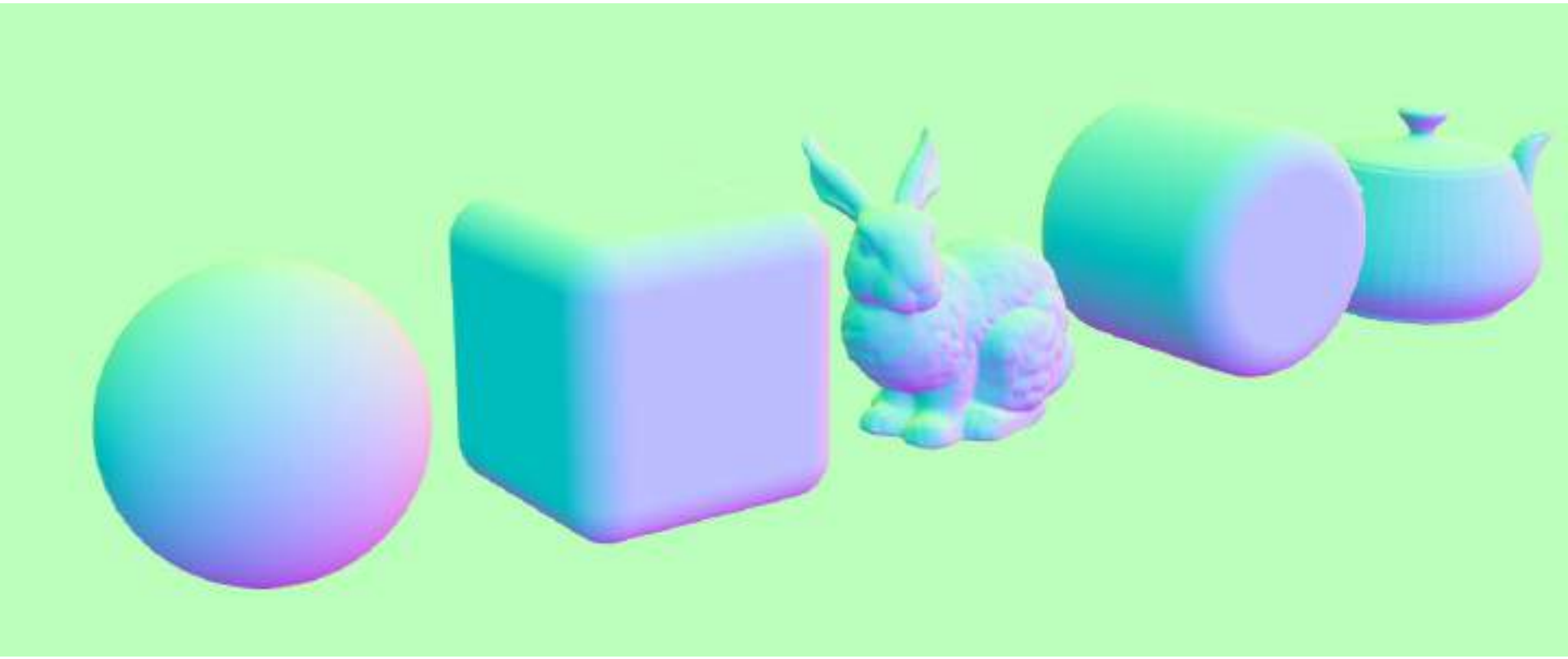
- Can make a small preview

```
vec3f shade_normal(scene* scene, ray3f ray) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return zero3f;
  auto object = scn->objects[isec.object];
  auto normal = transform_direction(object->frame,
    evalute_normal(object->shape, isec.element, isec.uv));
  return normal * 0.5 + 0.5;
}
```


© 2020 Fabio Pellacini

Normal shading

- Can make a small preview



© 2020 Fabio Pellacini

Diffuse illumination

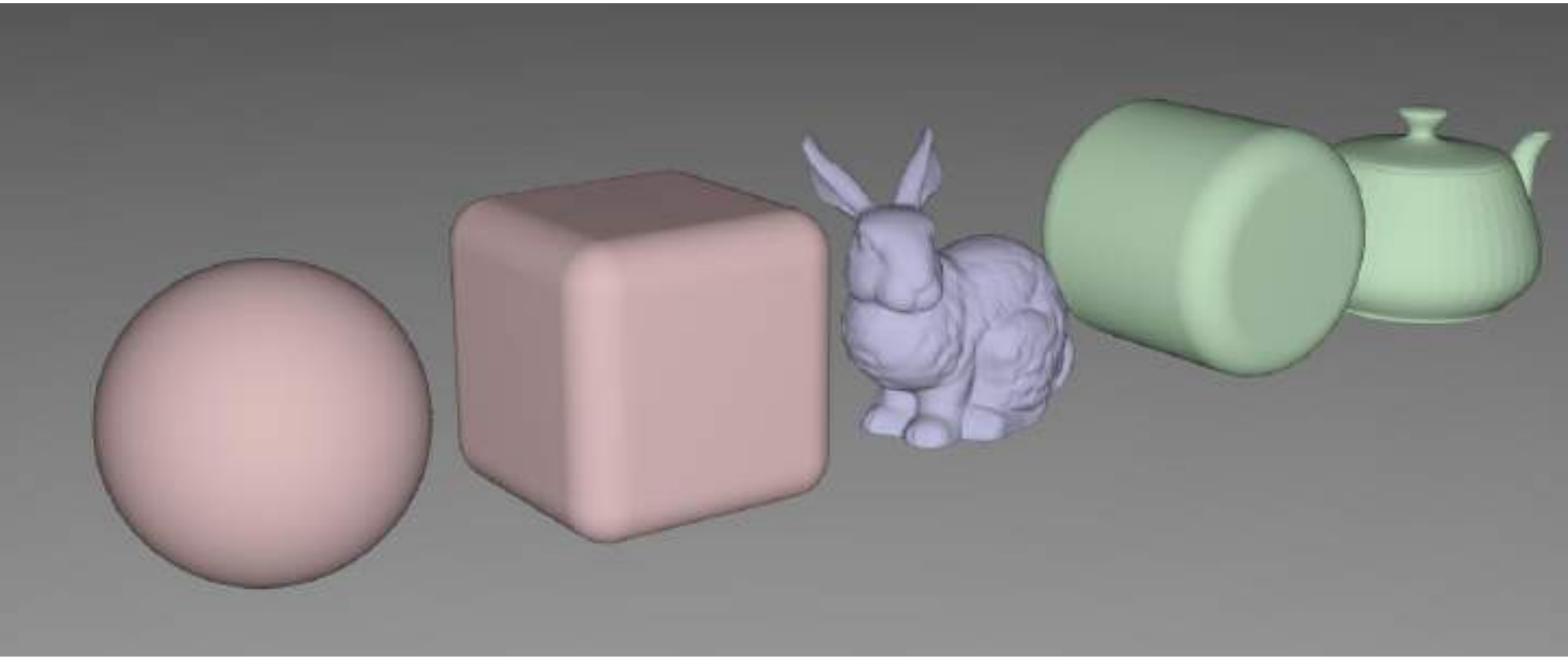
- Then, make a simple preview by pretending there is a light at the camera position, and avoid light falloff and normalization

```
vec3f shade_eyelight(scene* scene, ray3f ray) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return zero3f;
  auto object = scn->objects[isec.object];
  auto normal = transform_direction(object->frame,
    evalute_normal(object->shape, isec.element, isec.uv));
  return object->material->color * dot(normal, -ray.d);
}
```


© 2020 Fabio Pellacini

Diffuse illumination

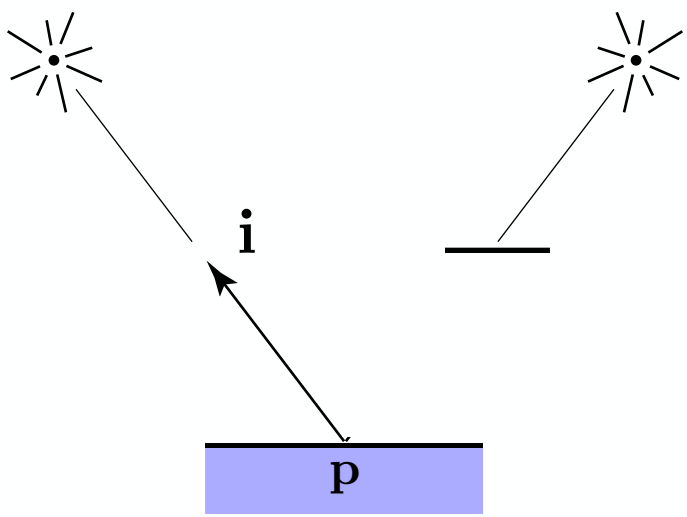
- Then, make a simple preview by pretending there is a list at the camera position, and avoid light falloff



© 2020 Fabio Pellacini

Shadows

- Surface is only illuminated if nothing blocks its view of the light
- Check for that by casting a ray toward the light



visibility
of source

$$L_d = \sum_i \frac{k_d}{\pi} \frac{I_i}{|\mathbf{s}_i - \mathbf{p}|^2} V(\mathbf{s}_i, \mathbf{p}) |\mathbf{n} \cdot \mathbf{i}_i|$$

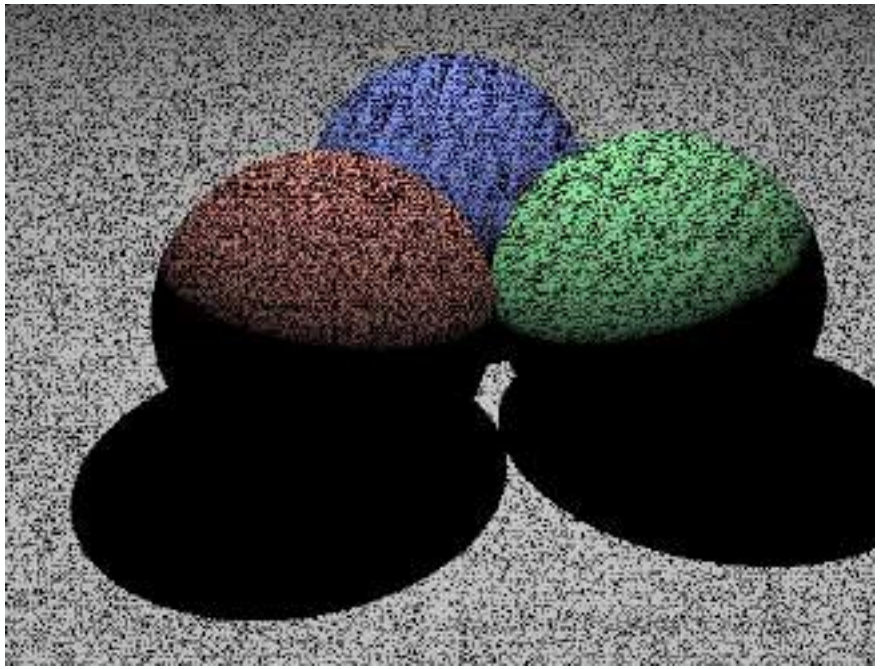
$$V(\mathbf{s}, \mathbf{p}) = !\mathrm{intersect}(ray(\mathbf{p}, \mathbf{i}, \epsilon, |\mathbf{p} - \mathbf{s}|))$$

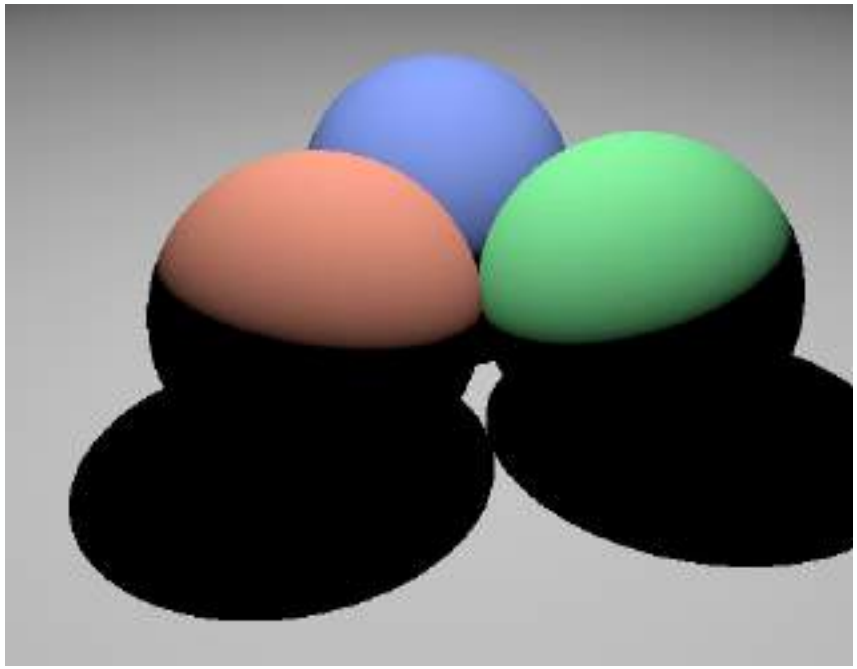
© 2020 Fabio Pellacini

Shadow rounding errors

- The ray may intersect the shape at the shading point
- Add a small epsilon to avoid this

```
auto sr = ray3f{p,l,eps,r};
```

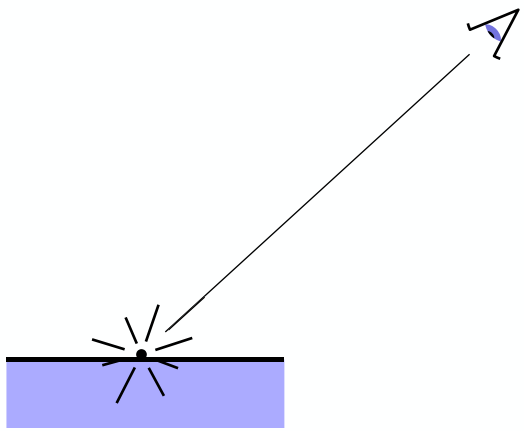




© 2020 Fabio Pellacini

Emission

- Surface not only reflect light, but also emit it
- We can simply add a constant term for this



$$L_e = k_e$$

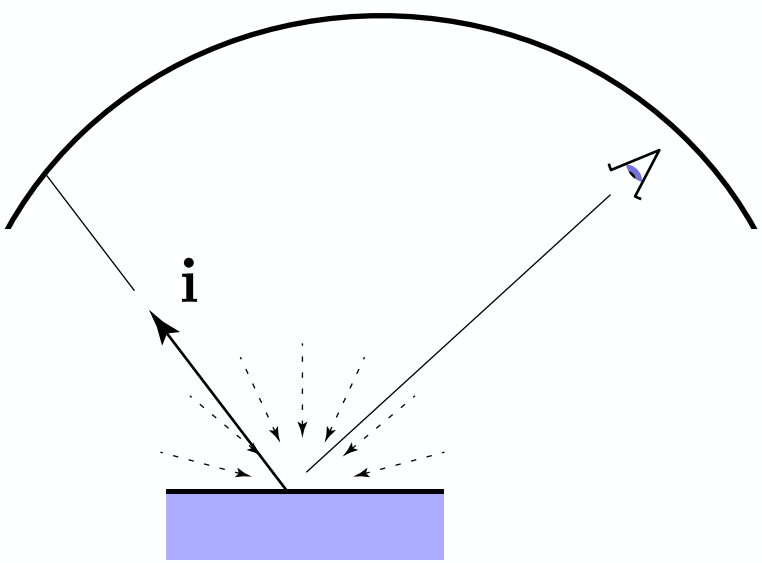
emission coefficient

emitted light

© 2020 Fabio Pellacini

Environment illumination

- Illumination from the environment comes from all directions
- We approximate it by averaging the illumination of random directions
- More directions leads to better approximation



size of hemisphere

environment illumination

visibility of source

$$L_d \approx \frac{2\pi}{n} \sum_{i}^{n} \frac{k_d}{\pi} E_e(\mathbf{i}_i) V(\mathbf{p}, \mathbf{i}_i) |\mathbf{n} \cdot \mathbf{i}_i|$$

reflected illumination

average of *n* samples

© 2020 Fabio Pellacini

Picking random directions

- To generate directions over the hemisphere, we pick two random numbers and use them as the cylindrical coordinates of a direction
- Note the formula is given in local coordinates, so transform it to world with a basis generate from the normal

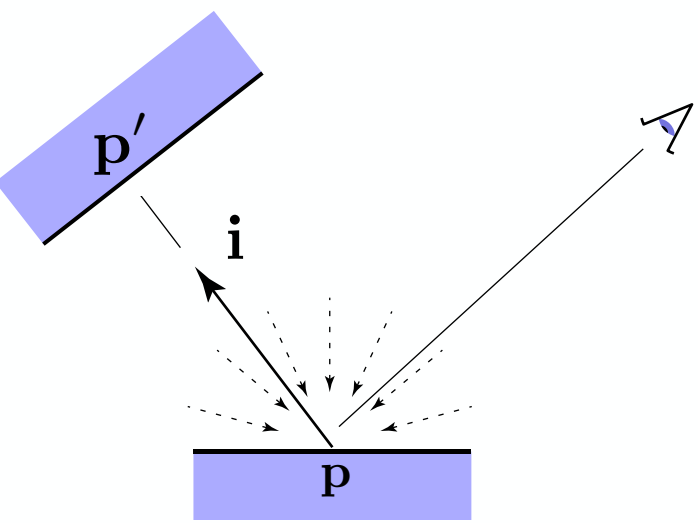
$$\mathbf{i}_l = \left(\sin(2\pi r_1)\sqrt{1-r_2^2}, \cos(2\pi r_1)\sqrt{1-r_2^2}, r_2\right)$$

$$\mathbf{i} = F_{\mathbf{n}} \cdot \mathbf{i}_l$$

© 2020 Fabio Pellacini

Indirect illumination

- Illumination comes also from all other objects
- As before, we can average the contribution from random directions
- We can determine the closely points via ray casting



emitted and reflected illumination from other points

$$\mathbf{p}'_i = \text{intersect}(ray(\mathbf{p}, \mathbf{i}_i))$$

$$L(\mathbf{p}) \approx L_e(\mathbf{p}) + \frac{2\pi}{n} \sum_i^n \frac{k_d}{\pi} L(\mathbf{p}'_i) |\mathbf{n} \cdot \mathbf{i}_i|$$

emitted and reflected illumination

emitted illumination

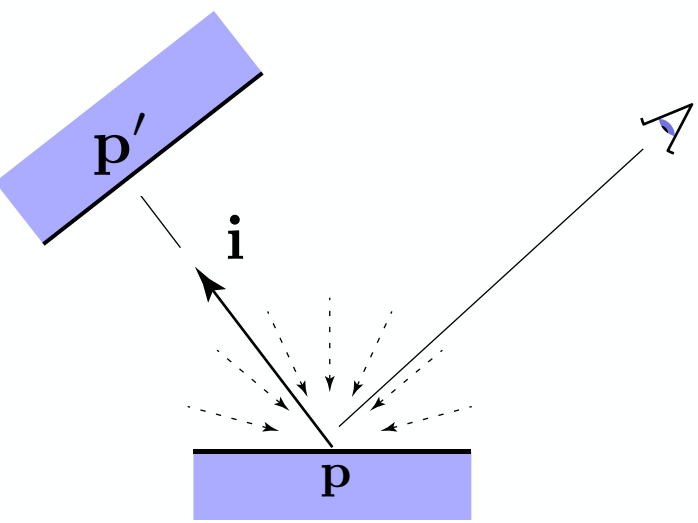
size of the hemisphere over samples

reflected illumination

© 2020 Fabio Pellacini

Indirect illumination

- Note how the formulation is recursive; a reasonable first approximation is to stop after 4-8 bounces
- Also, the number samples grows exponentially; we can simply pick only one sample and take the average in the "outer loop" shown later
- Finally, we can include environment illumination too by simply returning if if the ray does not hit a surface



$$L(\mathbf{p}) \approx L_e(\mathbf{p}) + \frac{2\pi}{n}\sum_i^n \frac{k_d}{\pi} L(\mathbf{p}'_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

emitted and reflected illumination

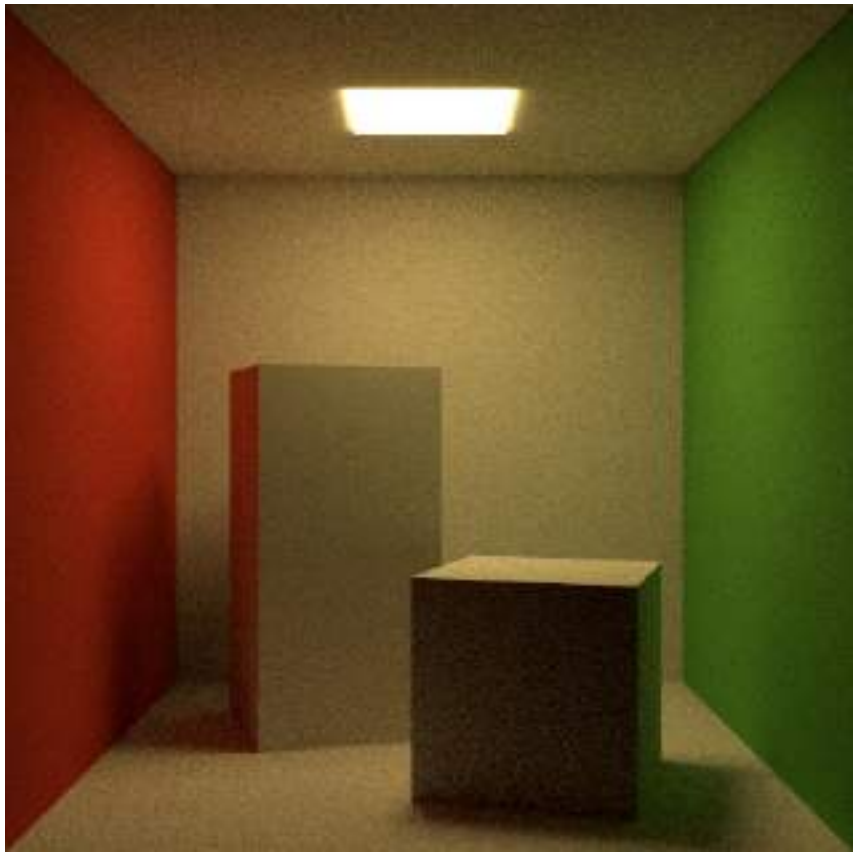
emitted illumination

reflected illumination

© 2020 Fabio Pellacini

Indirect illumination

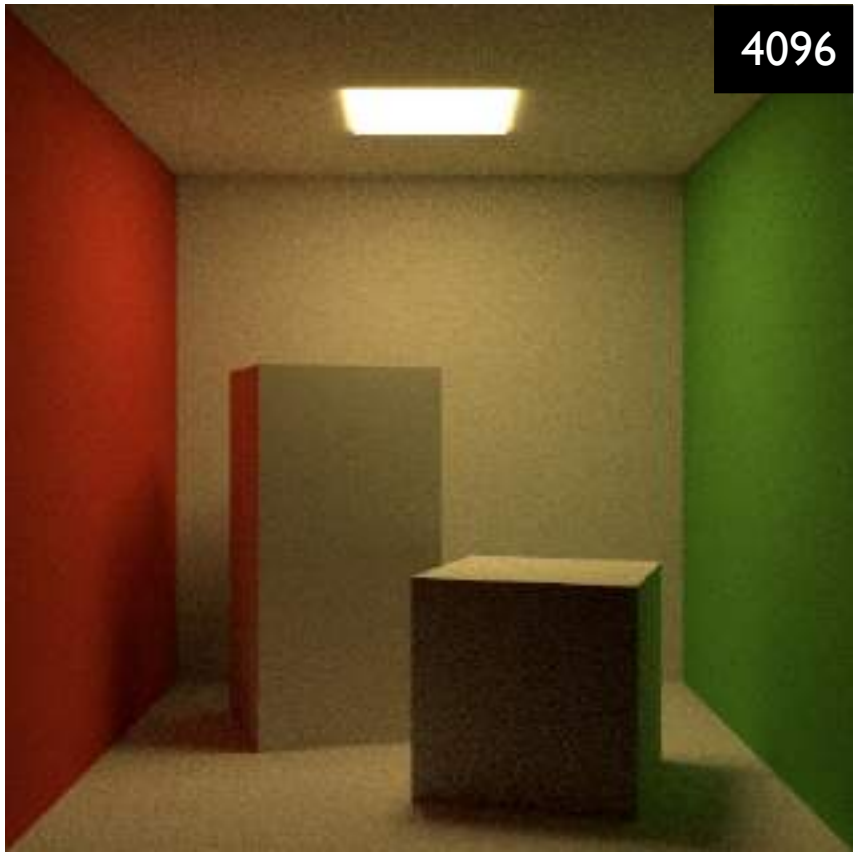
- Here is the effect of indirect illumination with an emissive surface



© 2020 Fabio Pellacini

Indirect illumination

- Here is the effect of indirect illumination with an emissive surface



© 2020 Fabio Pellacini

Illumination approximation

- Errors show up as noise: more samples means less noise

© 2020 Fabio Pellacini

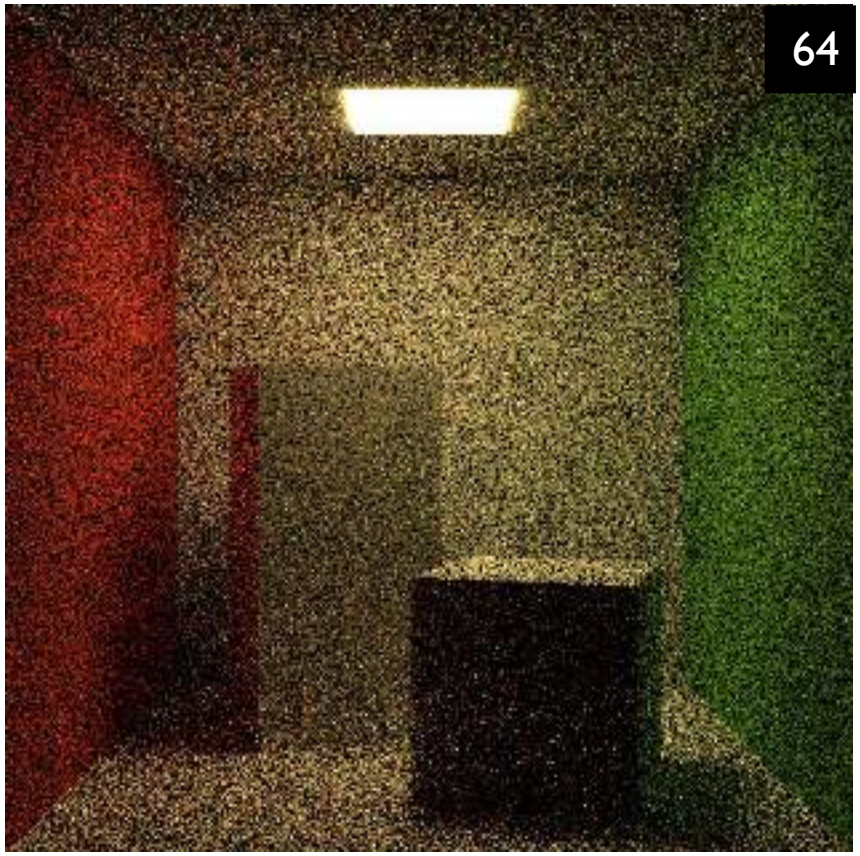
Illumination approximation

- Errors show up as noise: more samples means less noise

© 2020 Fabio Pellacini

Illumination approximation

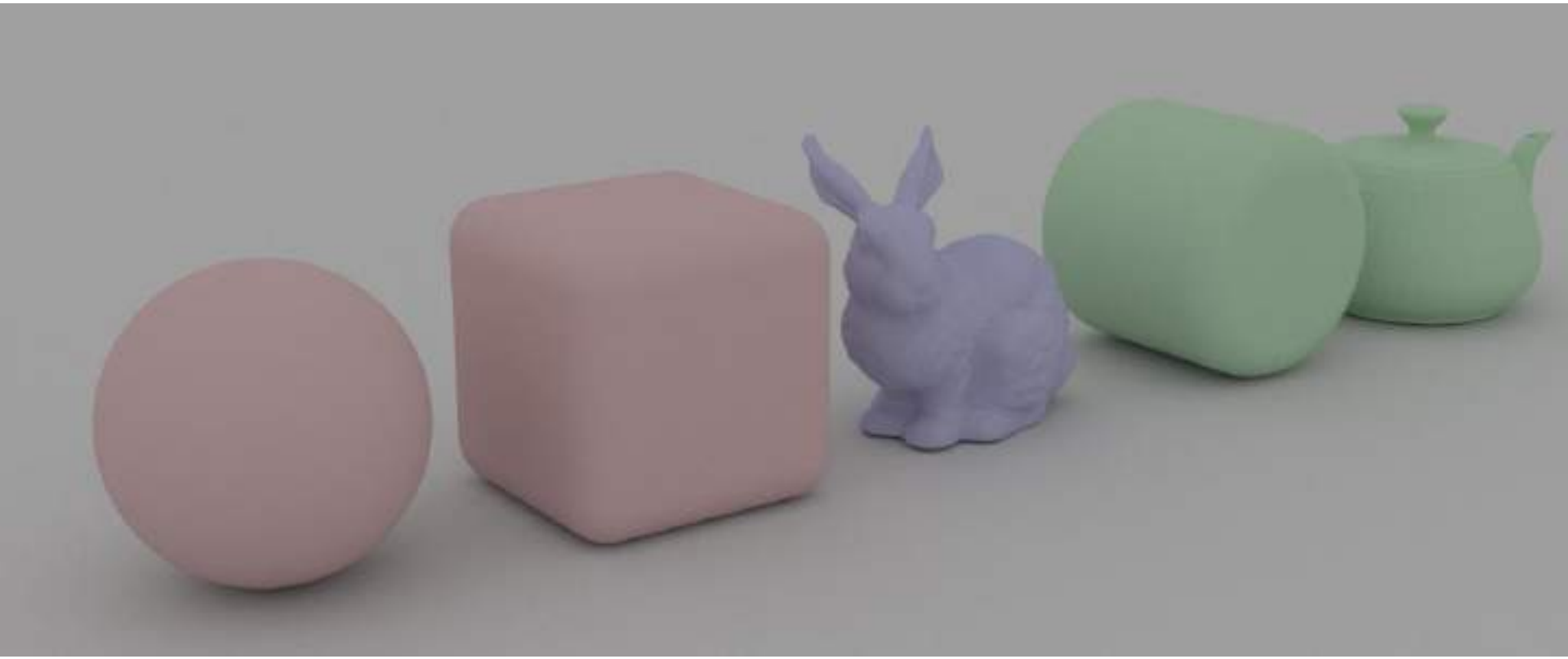
- Errors show up as noise: more samples means less noise



© 2020 Fabio Pellacini

Indirect and environment

- Shown here is the effect of indirect illumination with an environment



© 2020 Fabio Pellacini

Indirect illumination

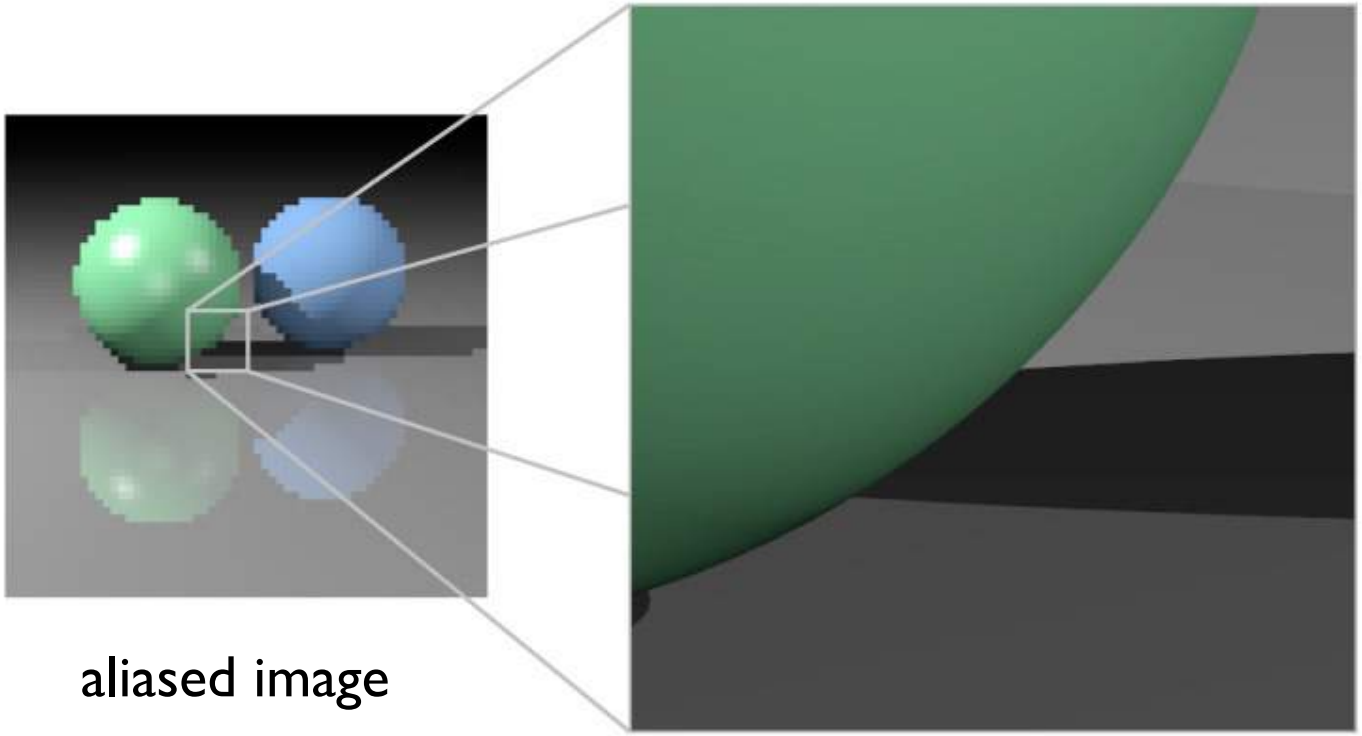
```
vec3f shade_indirect(scene* scene, ray3f ray, int bounce) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return eval_environment(scene, ray.d);
  auto object = scn->objects[isec.object];
  auto normal = transform_direction(object->frame,
    evalute_normal(object->shape, isec.element, isec.uv));
  auto position = transform_point(object->frame,
    evalute_position(object->shape,isec.element, isec.uv));
  auto radiance = object->material->emission;
  if(bounce >= max_bounce) return radiance;
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  radiance += (2 * pi) * object->material->color / pi *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
  return radiance;
}
```


© 2020 Fabio Pellacini

Antialiasing

© 2020 Fabio Pellacini

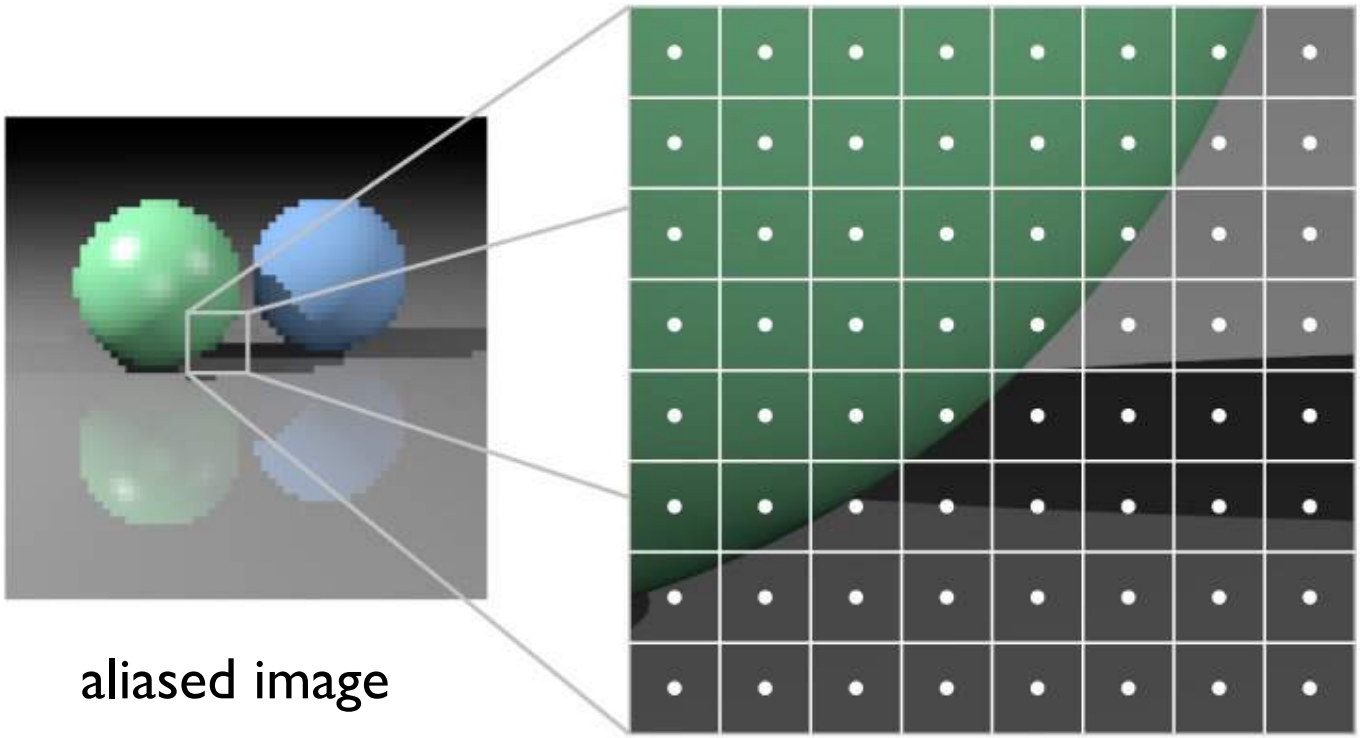
Antialiasing in ray tracing



aliased image

© 2020 Fabio Pellacini

Antialiasing in ray tracing

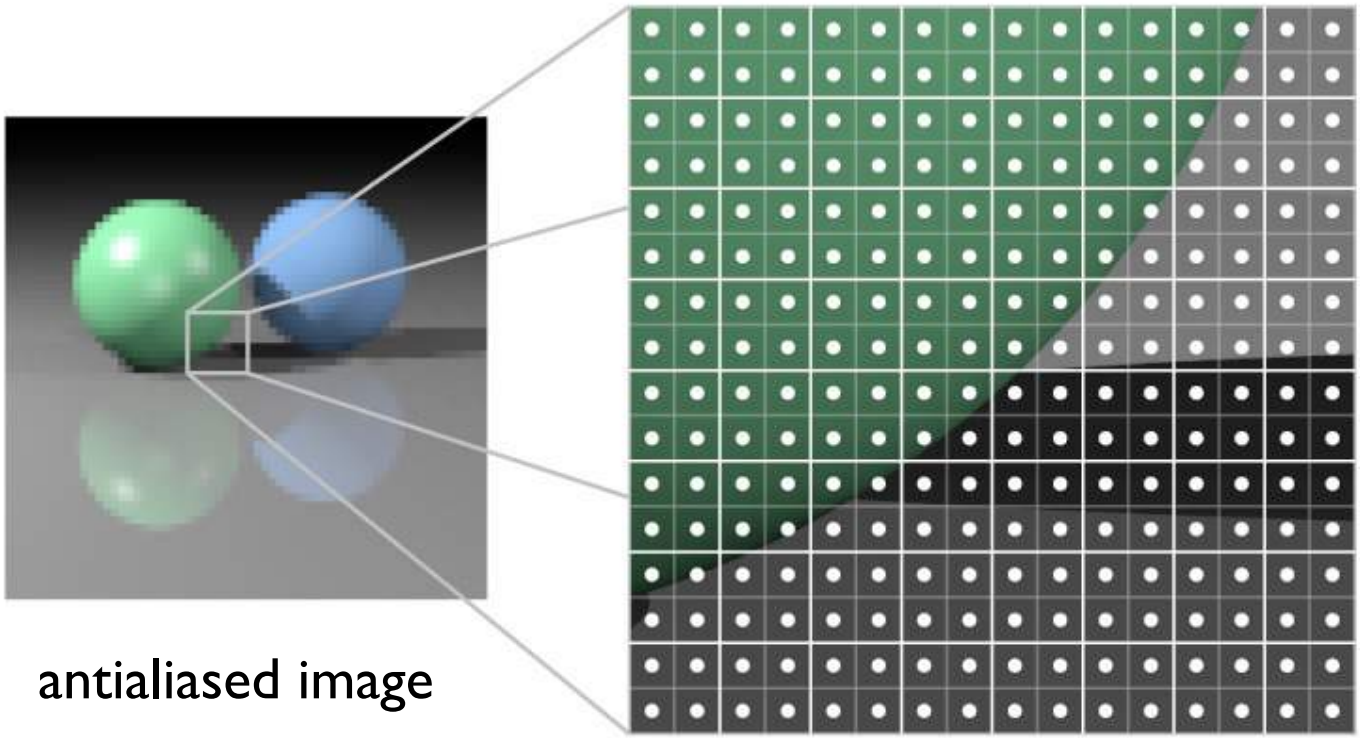


aliased image

one sample per pixel

© 2020 Fabio Pellacini

Antialiasing in ray tracing



antialiased image

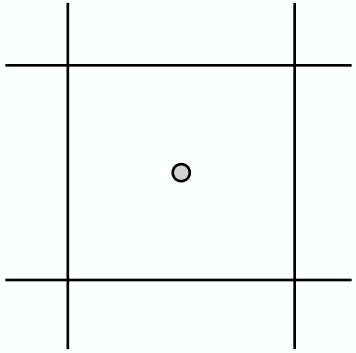
four samples per pixel

© 2020 Fabio Pellacini

Aliased raytracing

- Using only one sample per pixel introduces aliasing

```
// aliased raytracing
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    auto uv = (vec2f{i,j} + 0.5f) / img.size;
    auto ray = camera_ray(cam, uv);
    img[i,j] = shade(scene, ray);
  }
}
```

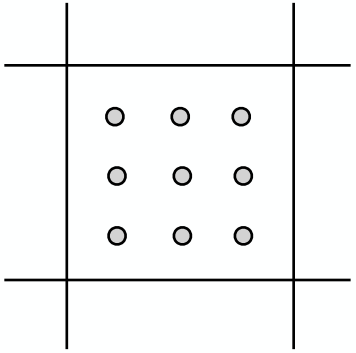


© 2020 Fabio Pellacini

Antialiasing by supersampling

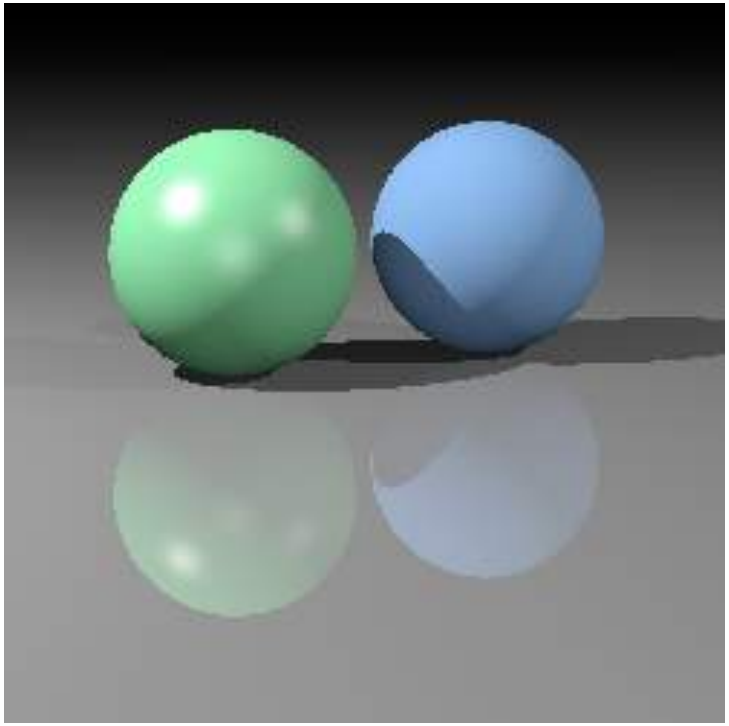
- Average over multiple points in each pixel to reduce aliasing

```
// antialiased with ns^2 samples per pixel
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    img[i,j] = {0,0,0};
    for(auto sj : range(ns)) {
      for(auto si : range(ns)) {
        auto puv = (vec2f{si,sj}+0.5f)/ns;
        auto uv = (vec2f{i,j} + puv) / img.size;
        auto ray = camera_ray(cam, uv);
        img[i,j] += shade(scene, ray);
      }
    }
    img[i,j] /= ns*ns;
  }
}
```

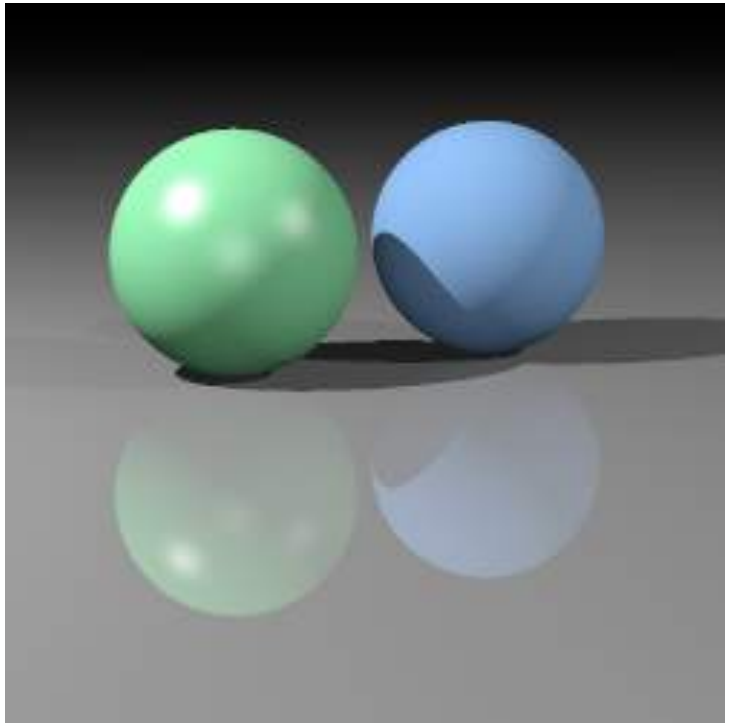


© 2020 Fabio Pellacini

Antialiasing in ray tracing



one sample per pixel



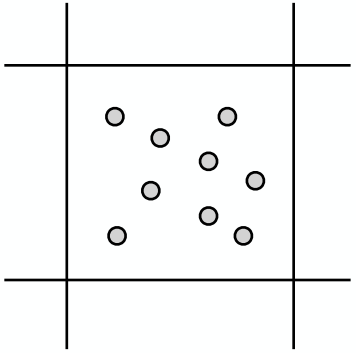
nine samples per pixel

© 2020 Fabio Pellacini

Antialiasing by random sampling

- We can also take locations at random in the pixel
- This allows to take any number of samples and can serve as outer loop for the shading functions given before

```
// antialiased with ns samples per pixel
auto rng = make_rng();
for(auto j : range(img.height)) {
  for(auto i : range(img.width)) {
    img[i,j] = {0,0,0};
    for(auto s : range(ns)) {
      auto puv = rand2f(rng);
      auto uv = (vec2f{i,j} + puv) / img.size;
      auto ray = camera_ray(cam, uv);
      img[i,j] += shade(scene, ray, rng);
    }
   img[i,j] /= ns;
  }
}
```

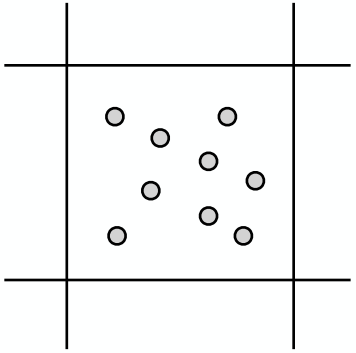


© 2020 Fabio Pellacini

Progressive rendering

- By inverting the loops, we can view images as they are computed
- For simplicity, we keep an accumulated image and rngs per pixels

```
// progressive with ns samples per pixel
auto rngs = make_rngs();
auto acc = image{img.size, zero3f};
for(auto s : range(ns)) {
  for(auto j : range(img.height)) {
    for(auto i : range(img.width)) {
      auto puv = rand2f(rngs[i,j]);
      auto uv = (vec2f{i,j} + puv) / img.size;
      auto ray = camera_ray(cam, uv);
      acc[i,j] += shade(scene, ray, rngs[i,j]);
      img[i,j] = acc[i,j] / (s + 1);
    }
  }
  // can view or save image here
}
```



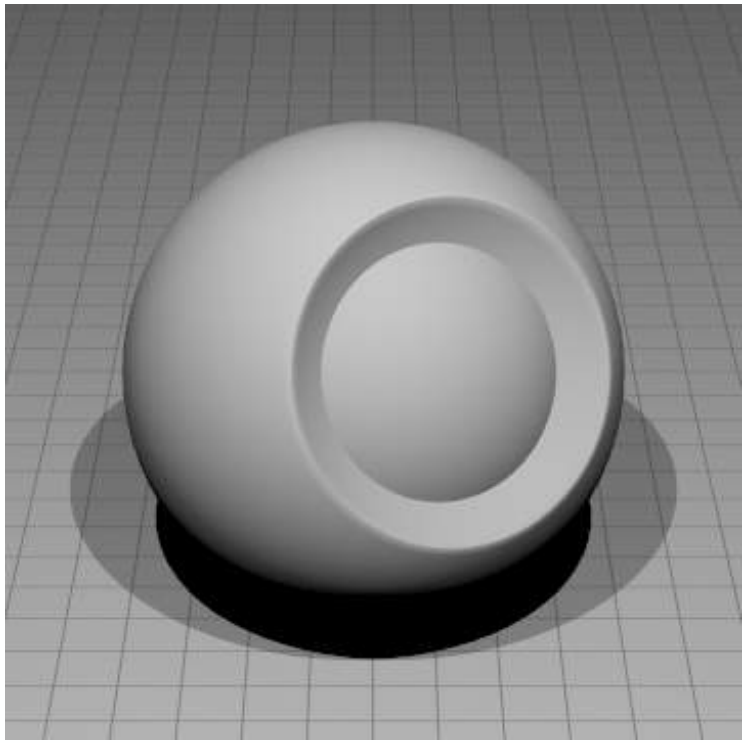
© 2020 Fabio Pellacini

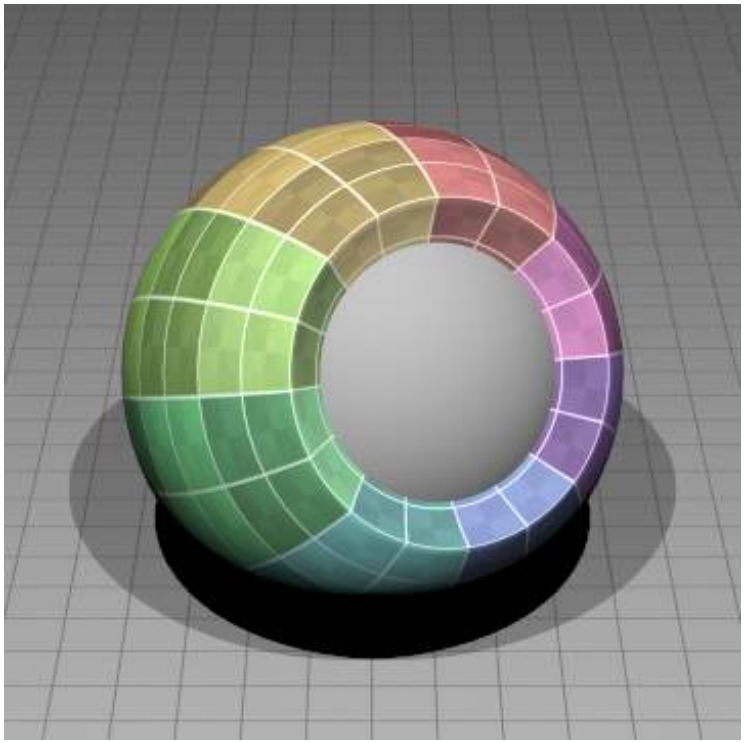
Textures

© 2020 Fabio Pellacini

Textures

- Render textures by modifying material parameters *kd* and *ks* in the lighting equation
  - typically by multiplying the texture colors by the coefficients



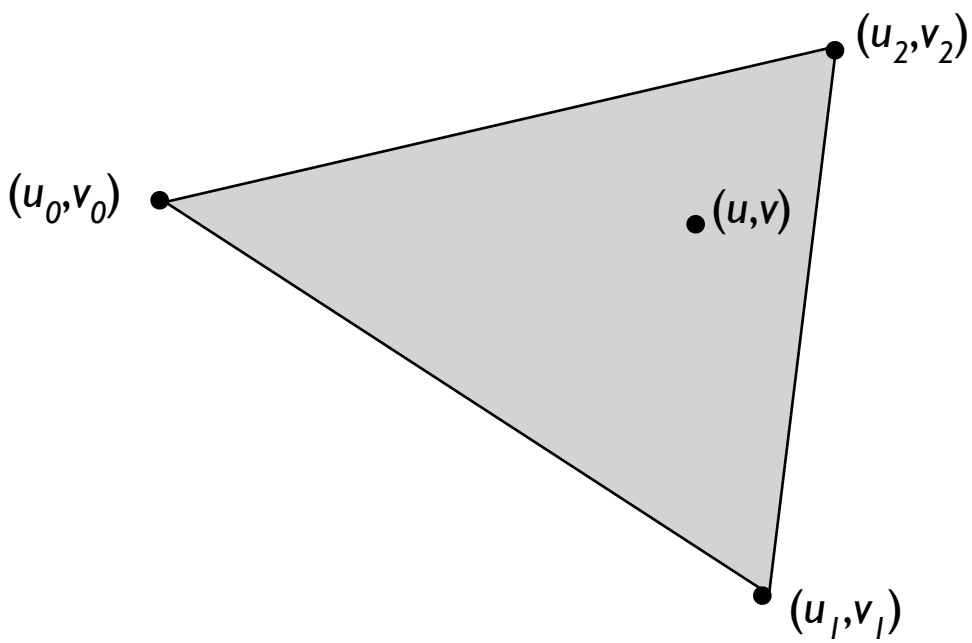


© 2020 Fabio Pellacini

Textures

- First, compute the texture coordinates *(u,v)* of the intersection point with barycentric interpolation of the intersected triangle

$$u = \sum w_i u_i \quad v = \sum w_i v_i$$



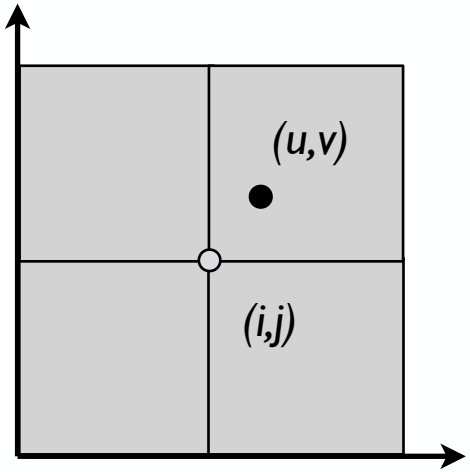
© 2020 Fabio Pellacini

Lookup Texture Values

- Then compute the image coordinate *(i,j)* for those *(u,v)*, taking into account tiling

$$s = (u \bmod 1) \cdot width \quad t = (v \bmod 1) \cdot height$$

$$i = \text{floor}(s) \quad j = \text{floor}(t)$$



© 2020 Fabio Pellacini

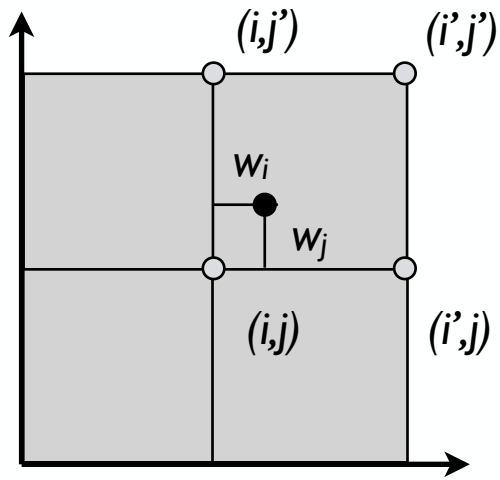
Lookup Texture Values

- Compute final value by interpolating between the closest 4 pixels; this avoids pixelation when we magnify the image

$$i' = (i+1) \bmod width \quad j' = (j+1) \bmod height$$

$$w_i = s - i \quad w_j = t - j$$

$$\mathbf{c} = (1-w_i)(1-w_j)\mathbf{c}_{ij} + w_i(1-w_j)\mathbf{c}_{i'j} + (1-w_i)w_j\mathbf{c}_{ij'} + w_i w_j \mathbf{c}_{i'j'}$$



© 2020 Fabio Pellacini

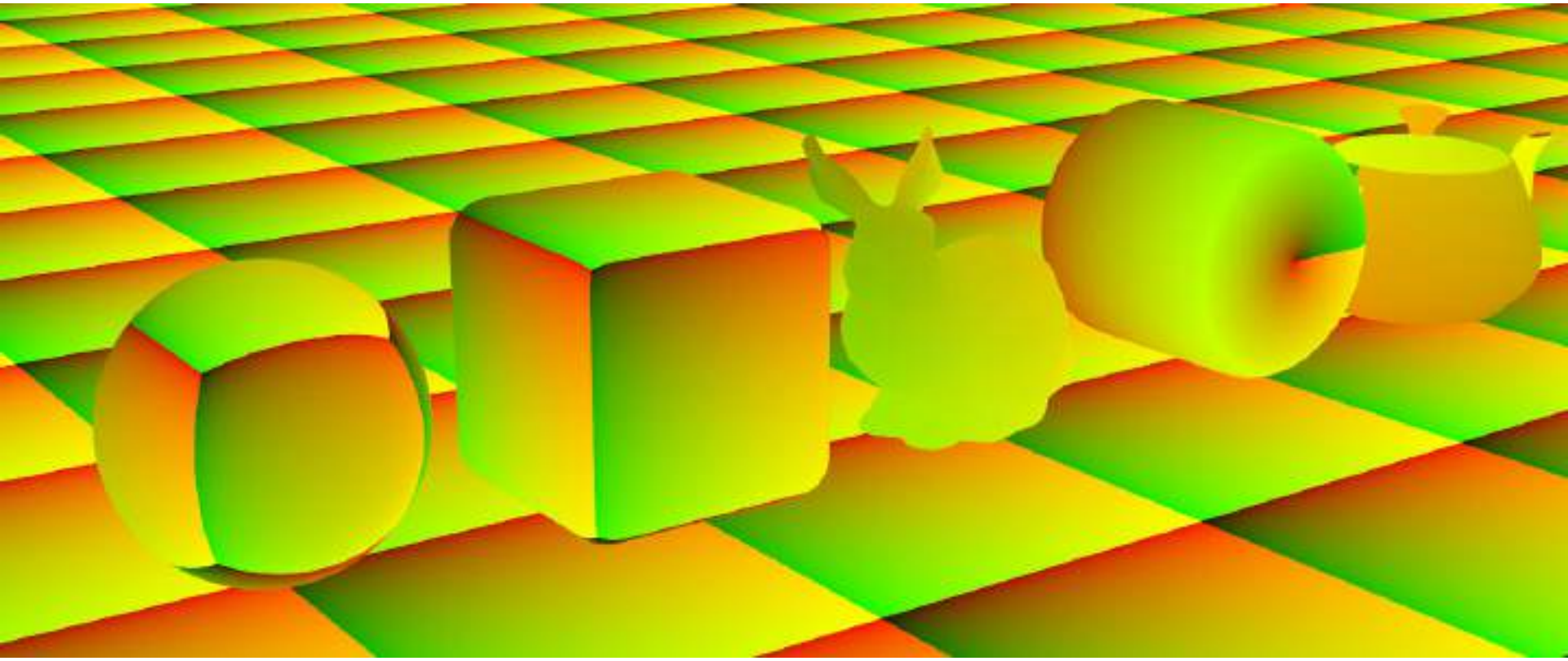
Lookup Texture Values

```
vec3f lookup(texture* texture, vec2i ij, bool no_srgb) {
  if(!texture->hdr.empty()) return texture->hdr[i,j];
  else if(no_srgb) return byte_to_float(texture->ldr[i,j])
  else return rgb_to_srgb(byte_to_float(texture->ldr[i,j]));
}

vec3f eval_texture(texture* texture, vec2f uv, bool no_srgb) {
  if (!texture) return {1, 1, 1};
  auto size = texture_size(texture);
  auto s = fmod(uv.x, 1) * size.x, t = fmod(uv.y, 1) * size.y;
  if (s < 0) s += size.x; if (t < 0) t += size.y;
  auto i=clamp((int)s,0,size.x-1), j=clamp((int)t,0,size.y-1);
  auto ii = (i + 1) % size.x, jj = (j + 1) % size.y;
  auto u = s - i, v = t - j;
  return lookup(texture,{ i,  j},no_srgb) * (1 - u) * (1 - v) +
         lookup(texture,{ i, jj},no_srgb) * (1 - u) * v +
         lookup(texture,{ii,  j},no_srgb) * u * (1 - v) +
         lookup(texture,{ii, jj},no_srgb) * u * v;
}
```


© 2020 Fabio Pellacini

Lookup Texture Values

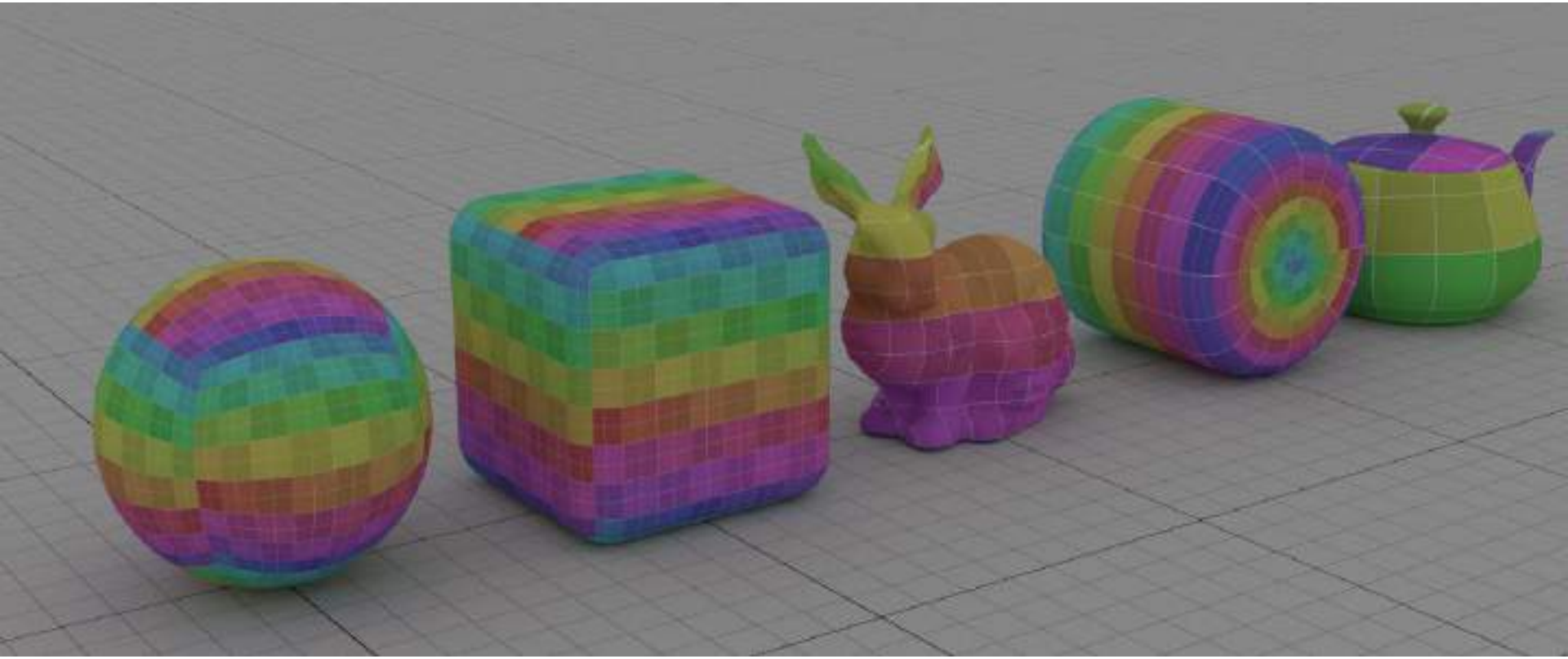
```
vec3f shade_texcoord(scene* scene, ray3f ray) {
 … return {fmod(eval_texcoord(shape, element, uv), 1), 0}; }
```



© 2020 Fabio Pellacini

Lookup Texture Values

```
vec3f shade_indirect(scene* scene, ray3f ray) {
 … auto color = material->color *
  eval_texture(material->color_tex); …}
```



© 2020 Fabio Pellacini

Lookup Environment Values

- Convert ray direction to latlong parametrization using spherical coordinates

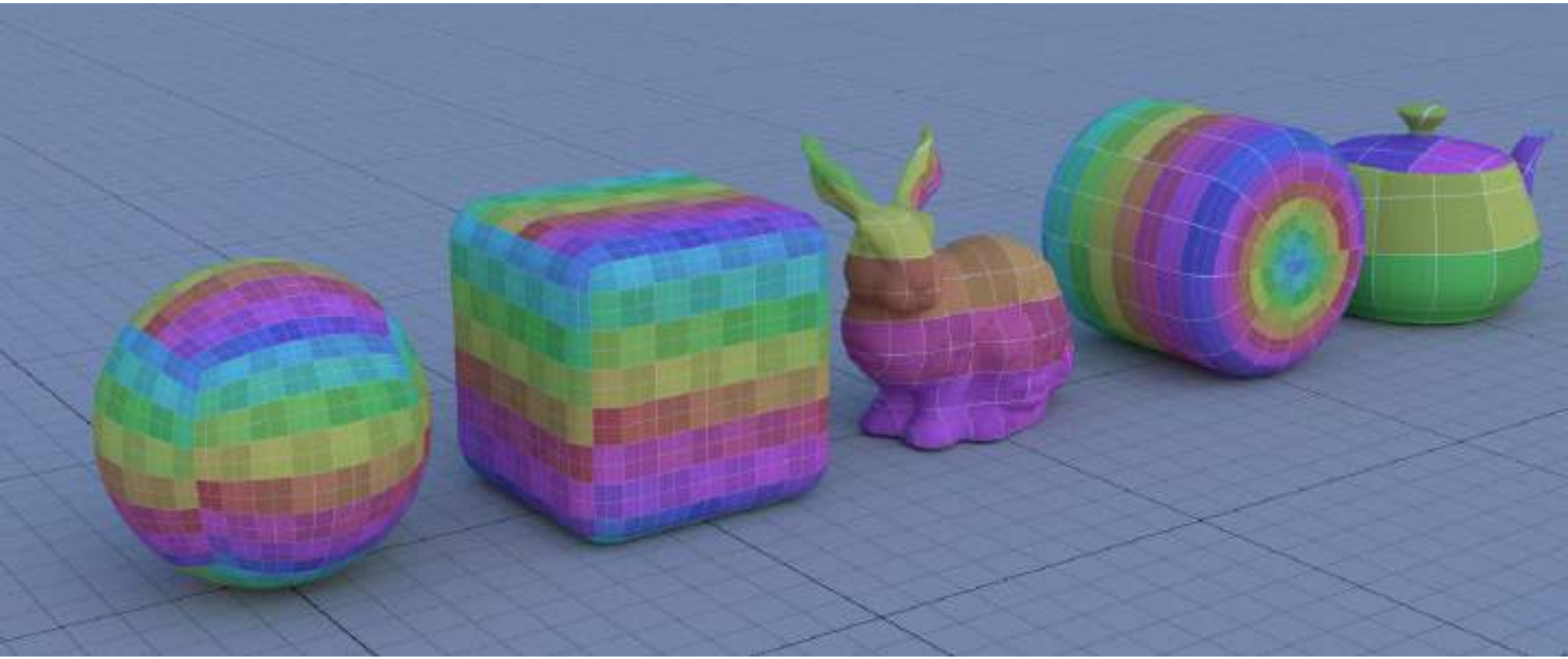
$$u = \frac{\mathrm{atan}(\mathbf{d}_z, \mathbf{d}_x)}{2\pi} \qquad v = \frac{\mathrm{acos}(\mathbf{d}_y)}{\pi}$$

```
vec3f eval_environment(environment* environment, vec3f dir) {
  auto local_dir = transform_direction(
                        inverse(environment->frame), dir)
  auto texcoord = vec2f{
        atan2(local_dir.z, local_dir.x) / (2 * pi),
        acos(clamp(local_dir.y, -1, 1)) / pi};
  if (texcoord.x < 0) texcoord.x += 1;
  return environment->emission *
            eval_texture(environment->emission_tex, texcoord);
}
```


© 2020 Fabio Pellacini

Lookup Environment Values

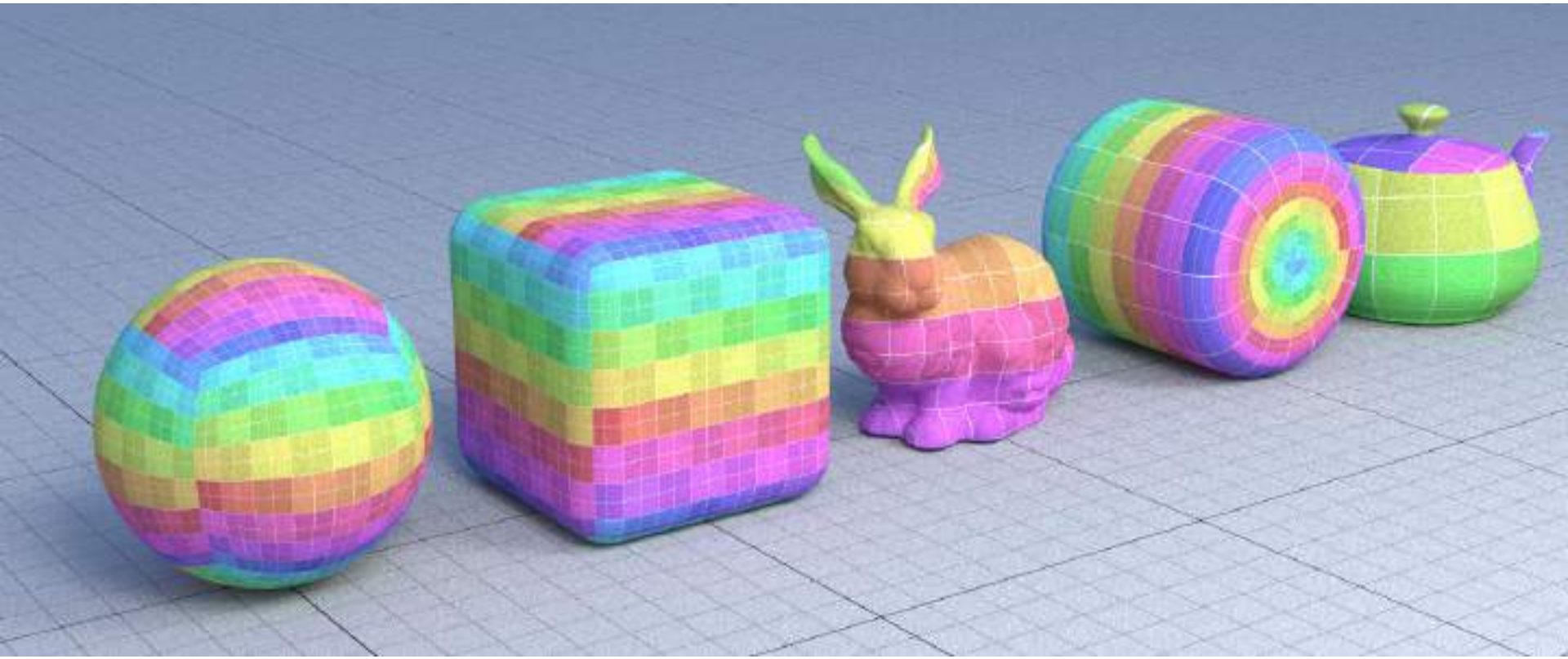
- Texture matte surfaces illuminated by a sky environment



© 2020 Fabio Pellacini

Lookup Environment Values

- Texture matte surfaces illuminated by a sky environment and area light

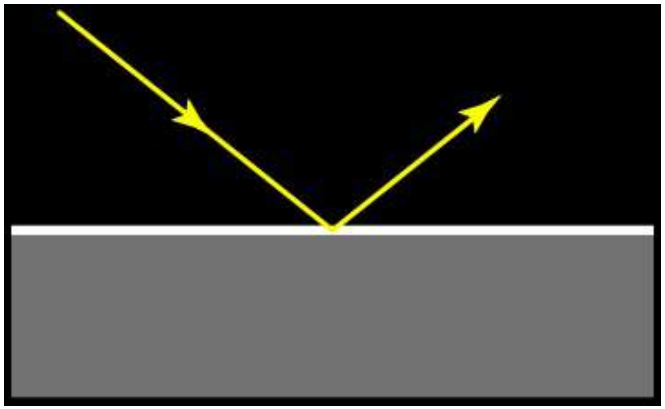


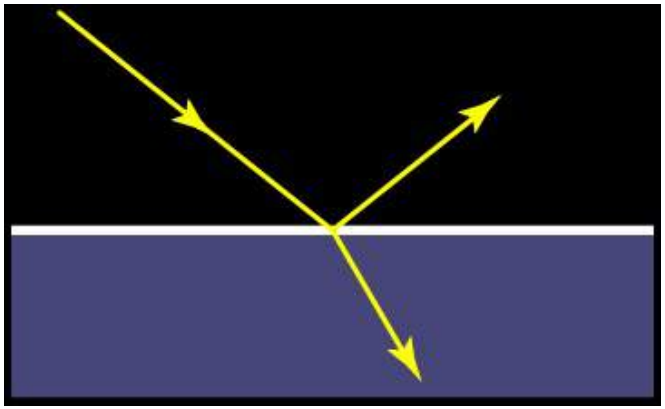
© 2020 Fabio Pellacini

Materials

© 2020 Fabio Pellacini

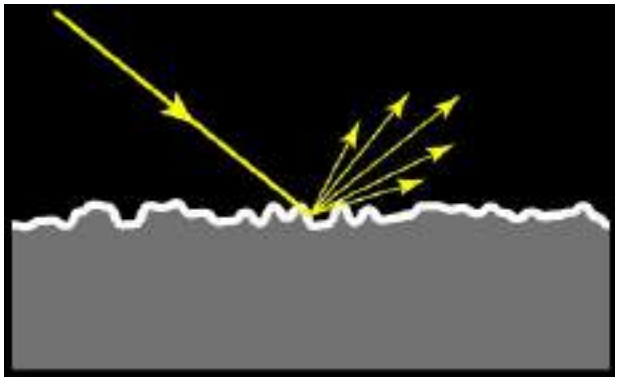
Polished surfaces

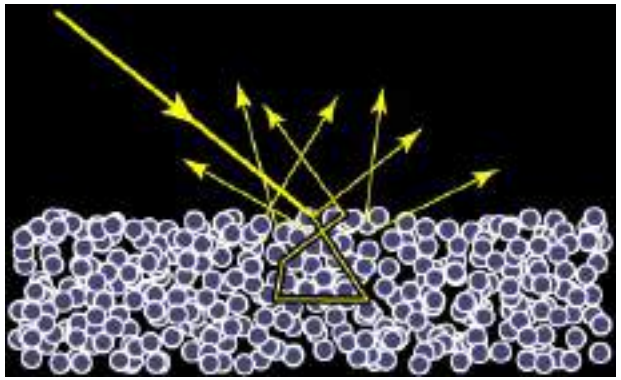




© 2017 Fabio Pellacini and Steve Marschner

Rough surfaces





© 2017 Fabio Pellacini and Steve Marschner

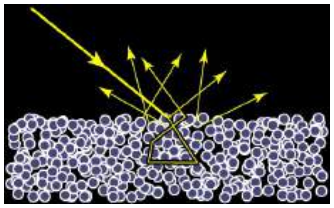
Handling multiple materials

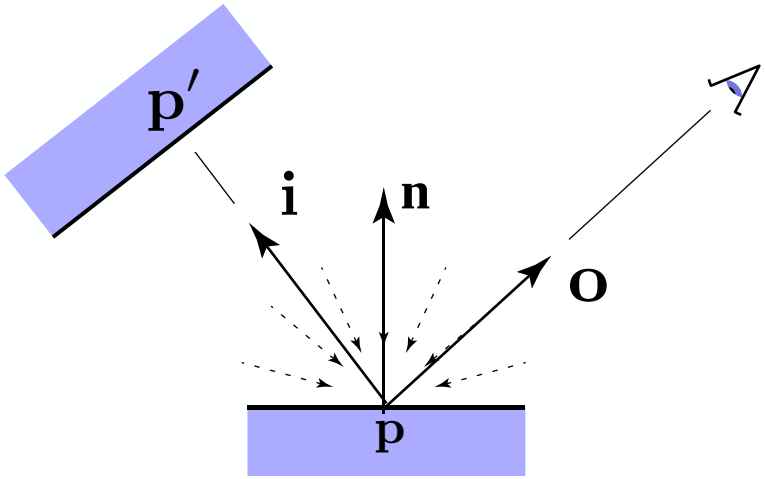
```
vec3f shade_indirect(scene* scene, ray3f ray, int bounce) {
  auto isec = intersection3f{};
  if(!intersect_scene(scene, ray, &isec))
    return eval_environment(scene, ray.d);
  <compute position, normal, texcoord>
  <compute material values accounting for textures>
  <handle opacity>
  auto radiance = object->material->emission;
  if(bounce >= max_bounce) return radiance;
  if (transmission) { <handle polished dielectrics> }
  else if (metallic && !roughness) { <handle polished metals> }
  else if (metallic && roughness) { <handle rough metals> }
  else if (specular) { <handle rough plastic> }
  else { <handle diffuse> }
  return radiance;
}
```


© 2020 Fabio Pellacini

Matte surfaces

- Matte surfaces scatter light in all directions
- They are approximated well by diffuse reflection
- We introduce a simplified notation to focus on material differences





$$L(\mathbf{p}, \mathbf{o}) \approx \frac{2\pi}{n} \sum_{i}^{n} \frac{k_d}{\pi} L(\mathbf{p}'_i, -\mathbf{i}_i) |\mathbf{n} \cdot \mathbf{i}_i|$$

$$\mathbf{p}'_i = \text{intersect}(ray(\mathbf{p}, \mathbf{i}_i))$$

© 2020 Fabio Pellacini

Matte surfaces

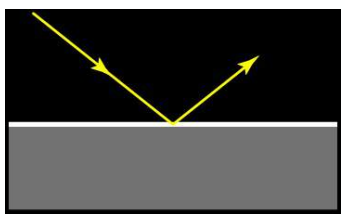
- Matte surfaces scatter light in all directions
- They are approximated well by diffuse reflection

```
// same as before
<handle diffuse> {
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  radiance += (2 * pi) * color / pi *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
}
```


© 2020 Fabio Pellacini

Polished mirrors

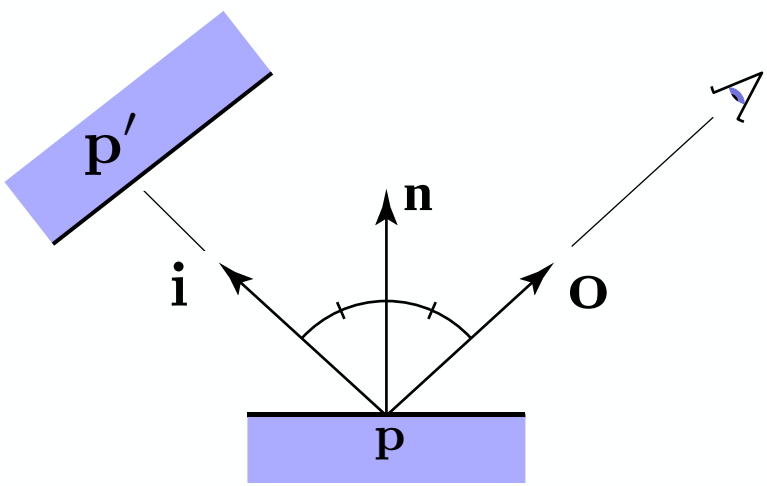
- Mirrors scatter light in exactly one direction
- They are approximated well by tracing a ray in the reflected direction



reflection coefficient

mirror direction

$$L(\mathbf{p},\mathbf{o}) \approx k_s L(\mathbf{p}'_r, -\mathbf{i}_r)$$



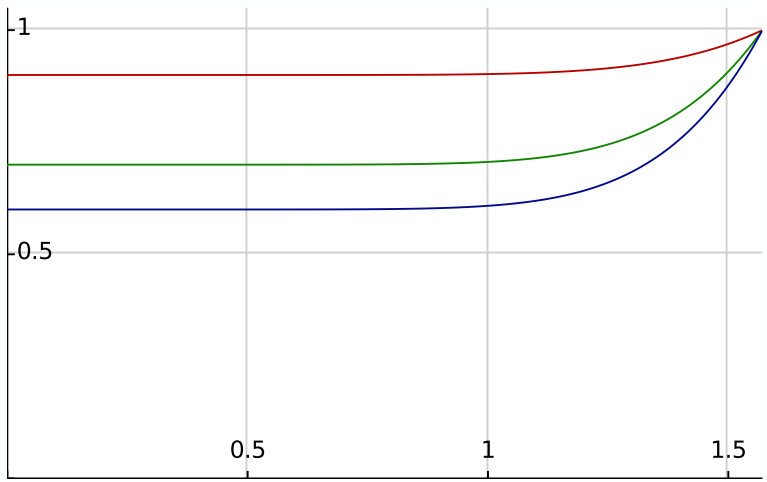
$$\mathbf{p}'_r = \text{intersect}(ray(\mathbf{p}, \mathbf{i}_r))$$

$$\mathbf{i}_r = -\mathbf{o} + 2(\mathbf{n}\cdot\mathbf{o})\mathbf{n}$$

© 2020 Fabio Pellacini

Polished metals

- Polished metals scatter light like mirrors
- The color of the surface though changes from the surface color at normal incidence to the color at grazing angle following Fresnel laws
- Here we show Schlick's approximation of the Fresnel term



$$L(\mathbf{p}, \mathbf{o}) \approx F(k_s, \mathbf{n}, \mathbf{o}) L(\mathbf{p}', -\mathbf{i})$$

$$\mathbf{p}'_r = \text{intersect}(ray(\mathbf{p}, \mathbf{i}_r))$$

$$\mathbf{i}_r = -\mathbf{o} + 2(\mathbf{n} \cdot \mathbf{o})\mathbf{n}$$

$$F(k_s, \mathbf{n}, \mathbf{o}) = k_s + (1 - k_s)(1 - \mathbf{n} \cdot \mathbf{o})^5$$

© 2020 Fabio Pellacini

Polished metals

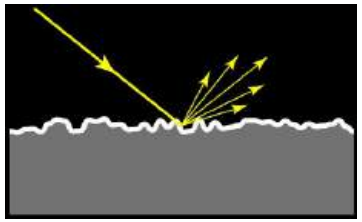
- Polished metals scatter light like mirrors
- The color of the surface though changes from the surface color at normal incidence to the color at grazing angle following Fresnel laws
- Here we show Schlick's approximation of the Fresnel term

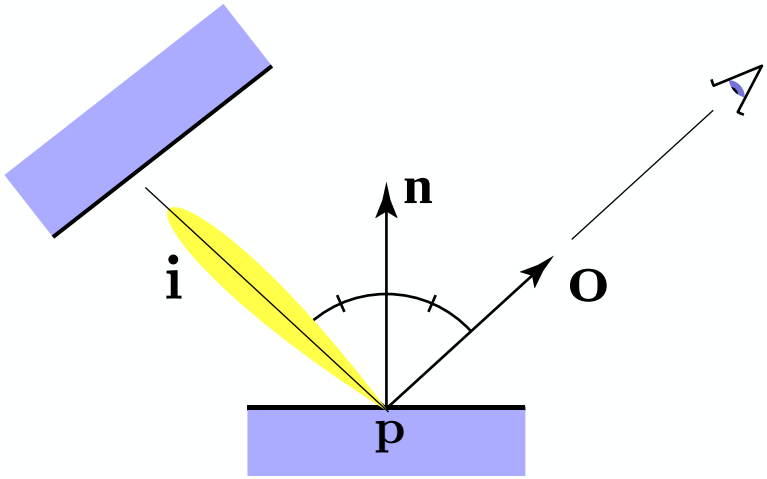
```
vec3f fresnel_schlick(vec3f ks, vec3f normal, vec3f outgoing) {
  return ks + (1 - ks)*pow(1 - dot(normal, outgoing), 5);
}

<handle polished metals> {
  auto incoming = reflect(outgoing, normal);
  radiance += fresnel_schlick(color, normal, outgoing)*
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1);
}
```

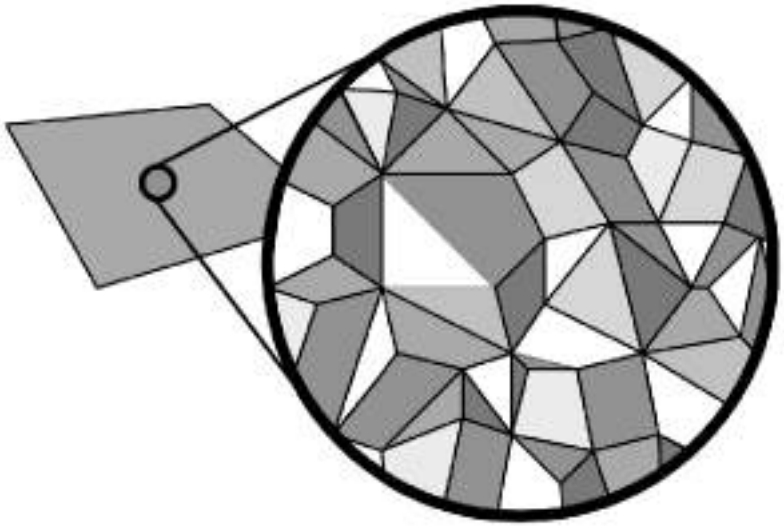

© 2020 Fabio Pellacini

Rough metals

- Rough metals scatter light around the reflected direction
- Their scattering is modeled by considering tiny micro-facets, each of which scatters light like a polished mirror







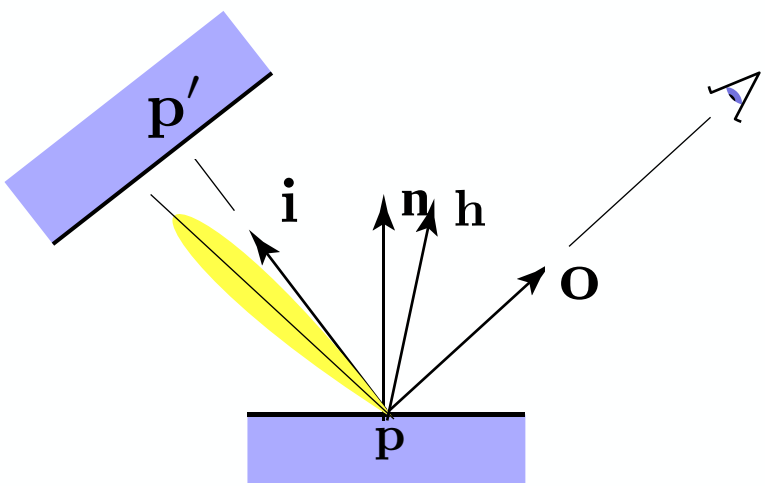
[Westin]

© 2020 Fabio Pellacini

Rough metals

- Rough metals scatter light around the reflected direction
- Their scattering is modeled by considering tiny micro-facets, each of which scatters light like a polished mirror

$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{F(\mathbf{h}_i,\mathbf{o})D(\mathbf{h}_i,\mathbf{n})G(\mathbf{n},\mathbf{o},\mathbf{i}_i)}{4|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}_i|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$



$$\mathbf{p}'_i = \mathrm{intersect}(ray(\mathbf{p},\mathbf{i}_i))$$

$$\mathbf{h} = \frac{\mathbf{o}+\mathbf{i}}{|\mathbf{o}+\mathbf{i}|}$$

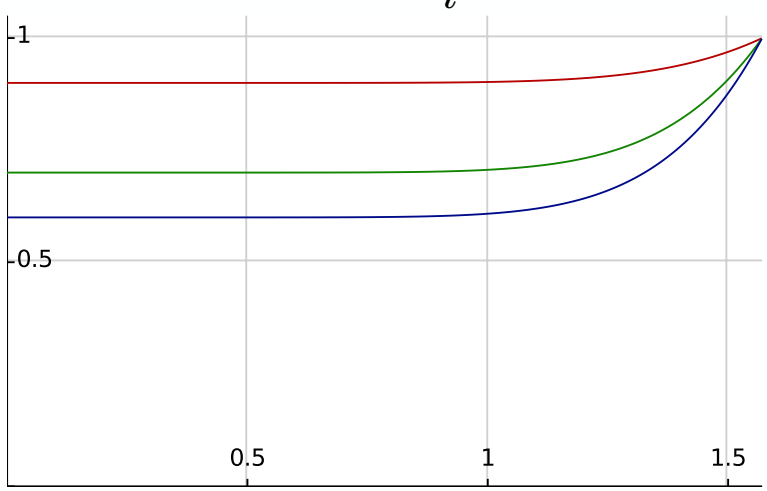
F: Fresnel term
D: microfacet distribution
G: microfacet shadowing

© 2020 Fabio Pellacini

Microfacet model

- Each micro facet scatters light like a metallic mirror
- Given an incoming and outgoing direction, the effective normal of the microfacet is the bisector *h* of the incoming *i* and outgoing o
- From this we evaluate the Fresnel term like before

$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{F(\mathbf{h}_i,\mathbf{o})D(\mathbf{h}_i,\mathbf{n})G(\mathbf{n},\mathbf{o},\mathbf{i}_i)}{4|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}_i|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$



$$\mathbf{h} = \frac{\mathbf{o}+\mathbf{i}}{|\mathbf{o}+\mathbf{i}|}$$

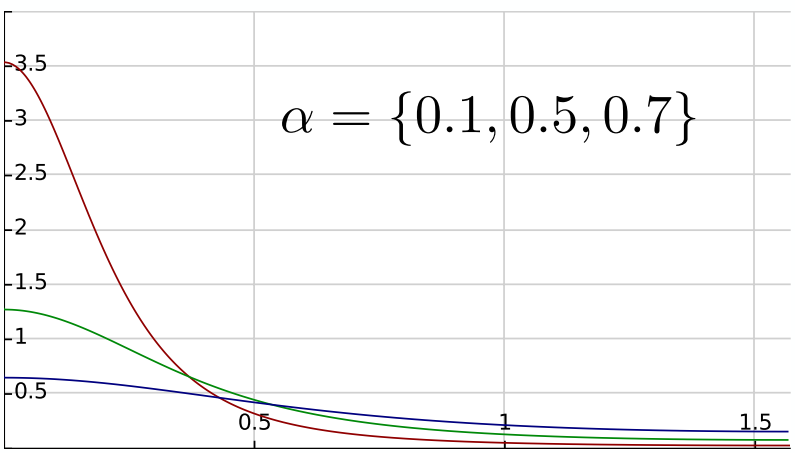
$$F(k_s,\mathbf{h},\mathbf{o}) = k_s + (1-k_s)(1-\mathbf{h}\cdot\mathbf{o})^5$$

© 2020 Fabio Pellacini

Microfacet distribution

- The microfacet distribution represent the ratio of microfacets oriented along the bisector *h*
- It depends on the normal and halfway directions and the surface roughness *a*, that measure how wide the distribution is

$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{F(\mathbf{h}_i,\mathbf{o})D(\mathbf{h}_i,\mathbf{n})G(\mathbf{n},\mathbf{o},\mathbf{i}_i)}{4|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}_i|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$



$$\mathbf{h} = \frac{\mathbf{o}+\mathbf{i}}{|\mathbf{o}+\mathbf{i}|}$$

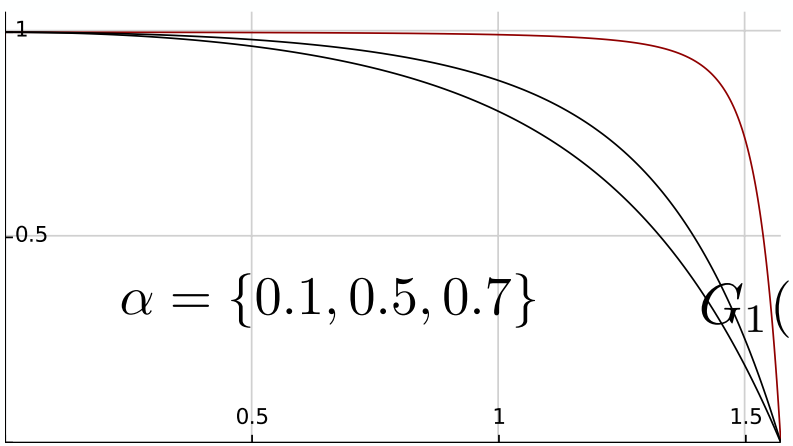
$$D(\mathbf{n},\mathbf{h},\alpha) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2-1)+1)^2}$$

© 2020 Fabio Pellacini

Microfacet distribution

- The shadowing term captures the ratio of microfacet that is visible from a particular incoming and outgoing angles
- It depends on the normal, incoming and outgoing directions and the surface roughness

$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\frac{F(\mathbf{h}_i,\mathbf{o})D(\mathbf{h}_i,\mathbf{n})G(\mathbf{n},\mathbf{o},\mathbf{i}_i)}{4|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}_i|}L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$



$$G(\mathbf{n},\mathbf{o},\mathbf{i},\alpha) = G_1(\mathbf{n},\mathbf{o},\alpha)G_1(\mathbf{n},\mathbf{i},\alpha)$$

$$G_1(\mathbf{n},\mathbf{o},\alpha) = \frac{2\,\mathbf{n}\cdot\mathbf{o}}{\mathbf{n}\cdot\mathbf{o}+\sqrt{\alpha^2+(1-\alpha^2)(\mathbf{n}\cdot\mathbf{o})^2}}$$

© 2020 Fabio Pellacini

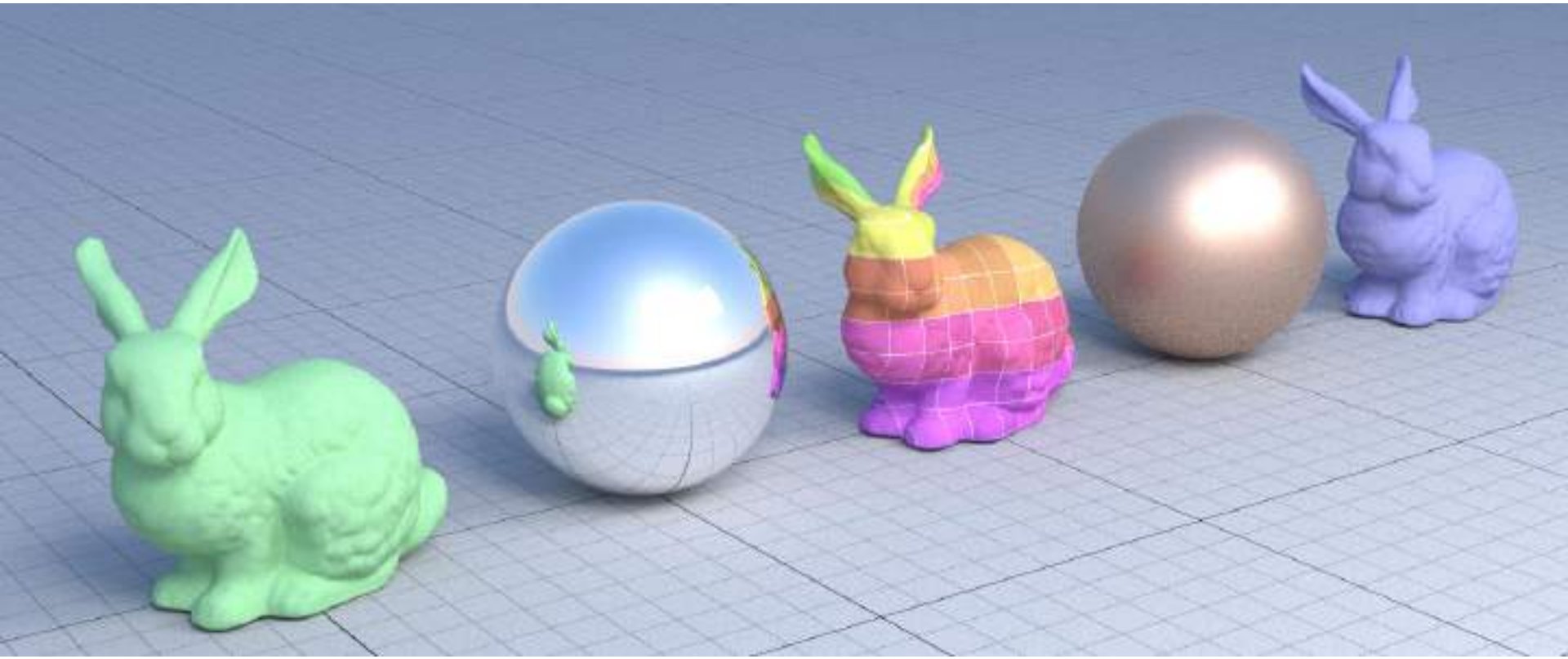
Rough metals

- Rough metals scatter light around the reflected direction
- Their scattering is modeled by considered tiny micro-facets, each of which scatters light like a polished mirror

```
float microfacet_distribution(…) { … }
float microfacet_shadowing(…) { … }

<handle rough metals> {
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  auto halfway = normalize(outgoing + incoming);
  radiance += (2 * pi) *
    fresnel_schlick(color, halfway, outgoing) *
    microfacet_distribution(roughness, normal, halfway) *
    microfacet_shadowing(roughness,normal,outgoing,incoming) /
    (4 * dot(normal, outgoing) * dot(normal, incoming)) *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
}
```


© 2020 Fabio Pellacini

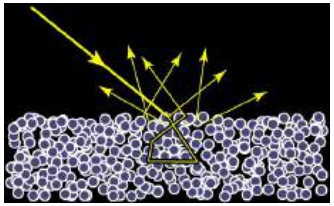
Polished and rough metals

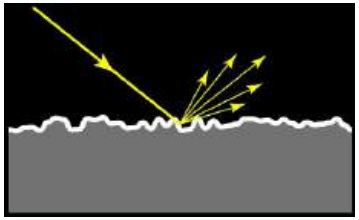


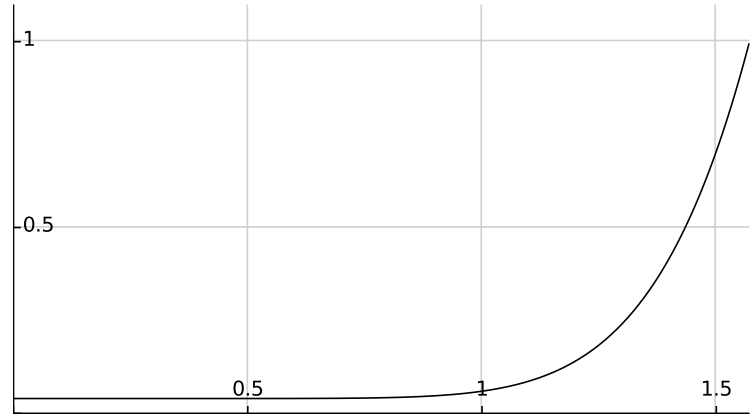
© 2020 Fabio Pellacini

Rough plastics

- Many surfaces can be modeled as a matte surface coated with a thin dielectric layer; an example of this are plastic surfaces
- They are approximated as sum of a diffuse and a specular contribution
- The specular layer reflects little light; k_s=0.04 is a good number for it
- The diffuse component is weighted by one minus the contribution of the specular one to conserve energy



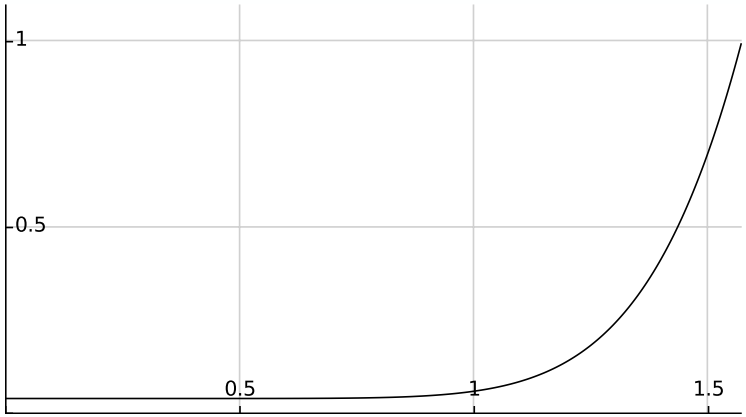




$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_i^n \left[\frac{k_d}{\pi}(1-F(\mathbf{h}_i,\mathbf{o})) + \frac{\overline{F(\mathbf{h}_i,\mathbf{o})}D(\mathbf{h}_i,\mathbf{n})G(\mathbf{n},\mathbf{o},\mathbf{i}_i)}{4|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}_i|}\right] L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

© 2020 Fabio Pellacini

Rough plastics



$$L(\mathbf{p},\mathbf{o}) \approx \frac{2\pi}{n}\sum_{i}^{n}\left[\frac{k_d}{\pi}(1-F(\mathbf{h}_i,\mathbf{o})) + \frac{F(\mathbf{h}_i,\mathbf{o})D(\mathbf{h}_i,\mathbf{n})G(\mathbf{n},\mathbf{o},\mathbf{i}_i)}{4|\mathbf{n}\cdot\mathbf{o}||\mathbf{n}\cdot\mathbf{i}_i|}\right] L(\mathbf{p}'_i,-\mathbf{i}_i)|\mathbf{n}\cdot\mathbf{i}_i|$$

© 2020 Fabio Pellacini

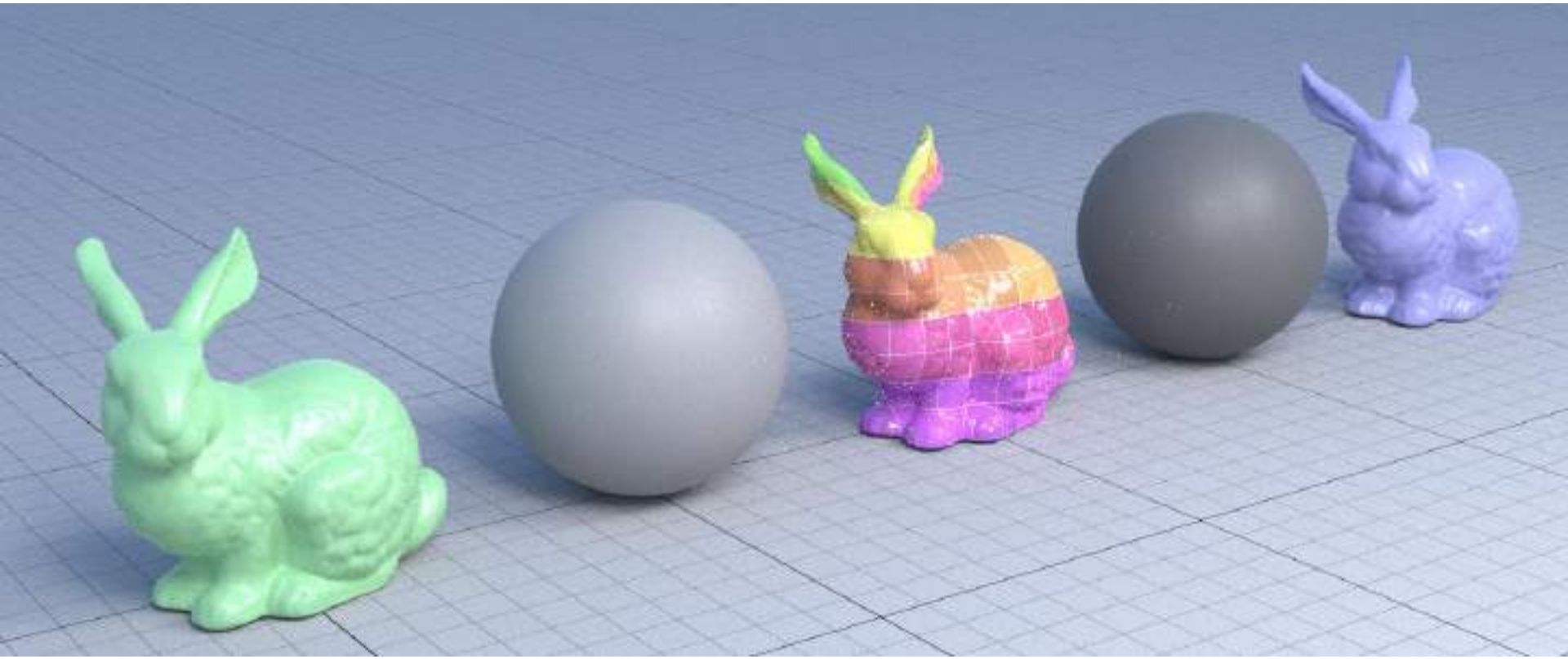
Rough plastics

- They are approximated as sum of a diffuse and a specular contribution

```
<handle rough plastic> {
  auto incoming = sample_hemisphere(normal, rand2f(rng));
  auto halfway = normalize(outgoing + incoming);
  radiance += (2 * pi) * (
    color / pi * (1 - fresnel_schlick(0.04,halfway,outgoing)) +
    fresnel_schlick(0.04, halfway, outgoing) *
    microfacet_distribution(roughness, normal, halfway) *
    microfacet_shadowing(roughness,normal,outgoing,incoming) /
    (4 * dot(normal, outgoing) * dot(normal, incoming))) *
    shade_indirect(scene, ray3f{position, incoming},
        bounce+1) * dot(normal, incoming);
}
```


© 2020 Fabio Pellacini

Rough plastics

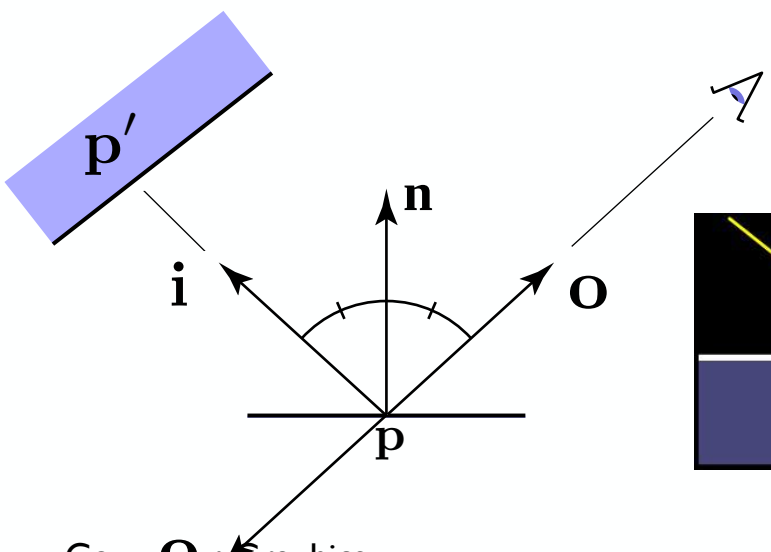


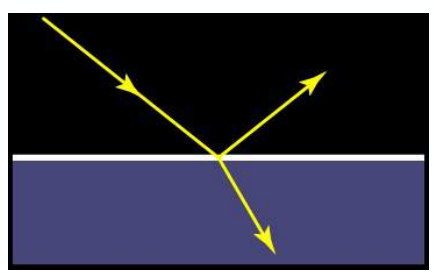
© 2020 Fabio Pellacini

Polished dielectrics

- Polished dielectrics scatter light both reflecting it and transmitting it
- We can model the reflection like mirrors but with a small reflection coefficient, k_s around 0.04
- The transmitted light is refracted into a material; for thin objects, we can cheaply approximate this by letting the ray continue and weighting it by one minus the fresnel term

$$L(\mathbf{p},\mathbf{o}) \approx F(k_s,\mathbf{n},\mathbf{o})L(\mathbf{p}'_r,-\mathbf{i}'_r)+$$
$$+(1-F(k_s,\mathbf{n},\mathbf{o}))L(\mathbf{p}'_t,-\mathbf{i}'_t)$$





$$\mathbf{i}_r = -\mathbf{o} + 2(\mathbf{n}\cdot\mathbf{o})\mathbf{n}$$

$$\mathbf{i}_t = -\mathbf{o}$$

© 2020 Fabio Pellacini

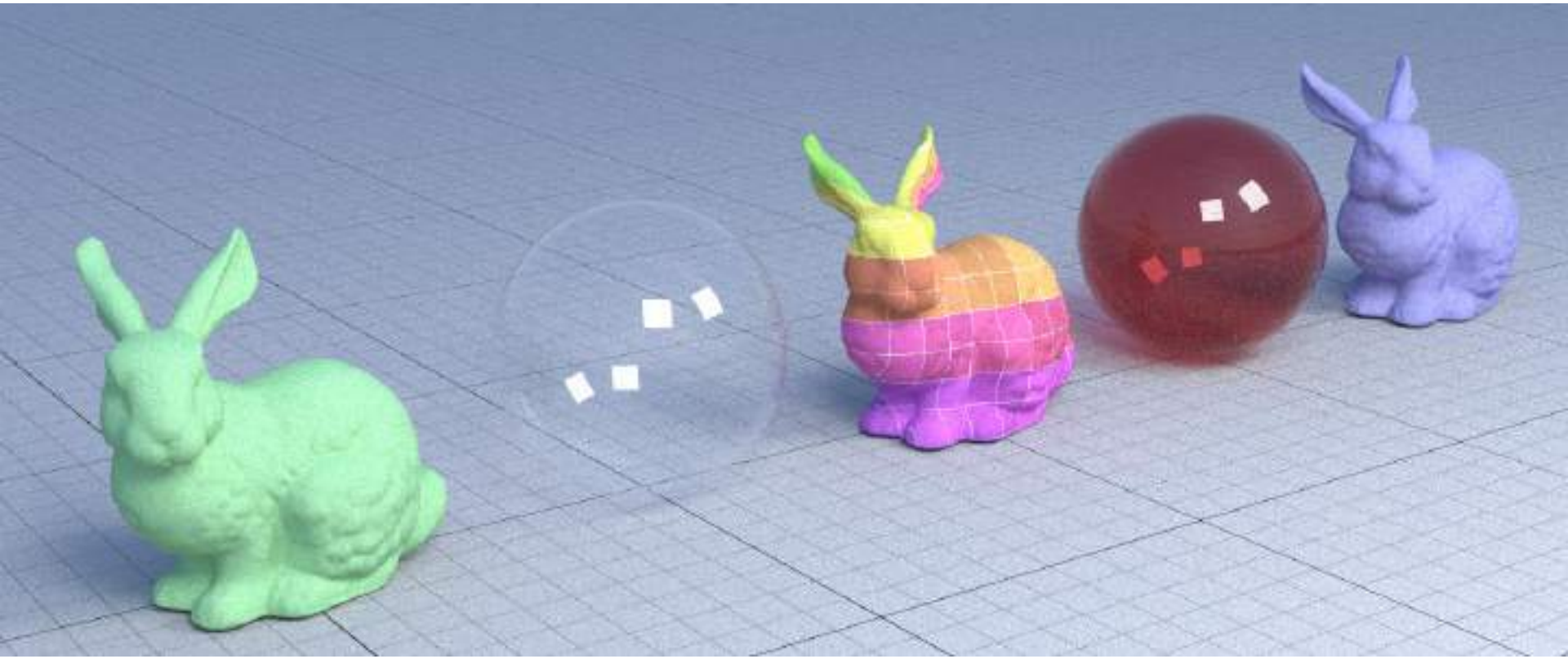
Polished dielectrics

- Polished dielectrics scatter light both reflecting it and transmitting it
- We want to have only one ray continue, so we randomly pick which direction to go based on the fresnel term
- Due to this random picking, we do not need to reweighs for the Fresnel term

```
<handle polished dielectrics> {
  if(rand1f(rng) < fresnel_schlick(0.04, normal, outgoing)) {
    auto incoming = reflect(outgoing, normal);
    radiance += shade_indirect(scene, ray3f{position, incoming},
          bounce+1);
  } else {
    auto incoming = -outgoing;
    radiance += color *
      shade_indirect(scene, ray3f{position, incoming},
          bounce+1);
  }
}
```


© 2020 Fabio Pellacini

Polished dielectrics

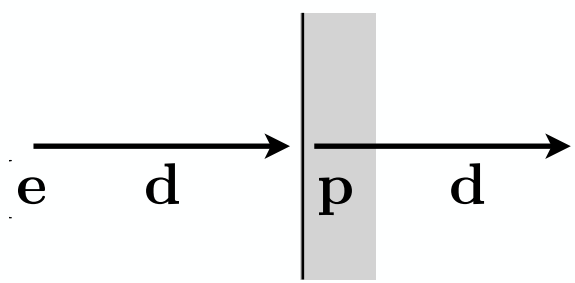


© 2020 Fabio Pellacini

Opacity

- As we saw for images, shapes can have an opacity stored in a texture
- This is helpful to model surfaces with holes without using triangles
- We will approximate this by continuing the ray is a random number is below the surface opacity

$$L_t = k_o \text{ raytrace}(\{\mathbf{e}, \mathbf{d}\}) + (1 - k_o) \text{ raytrace}(\{\mathbf{p}, \mathbf{d}\})$$



© 2020 Fabio Pellacini

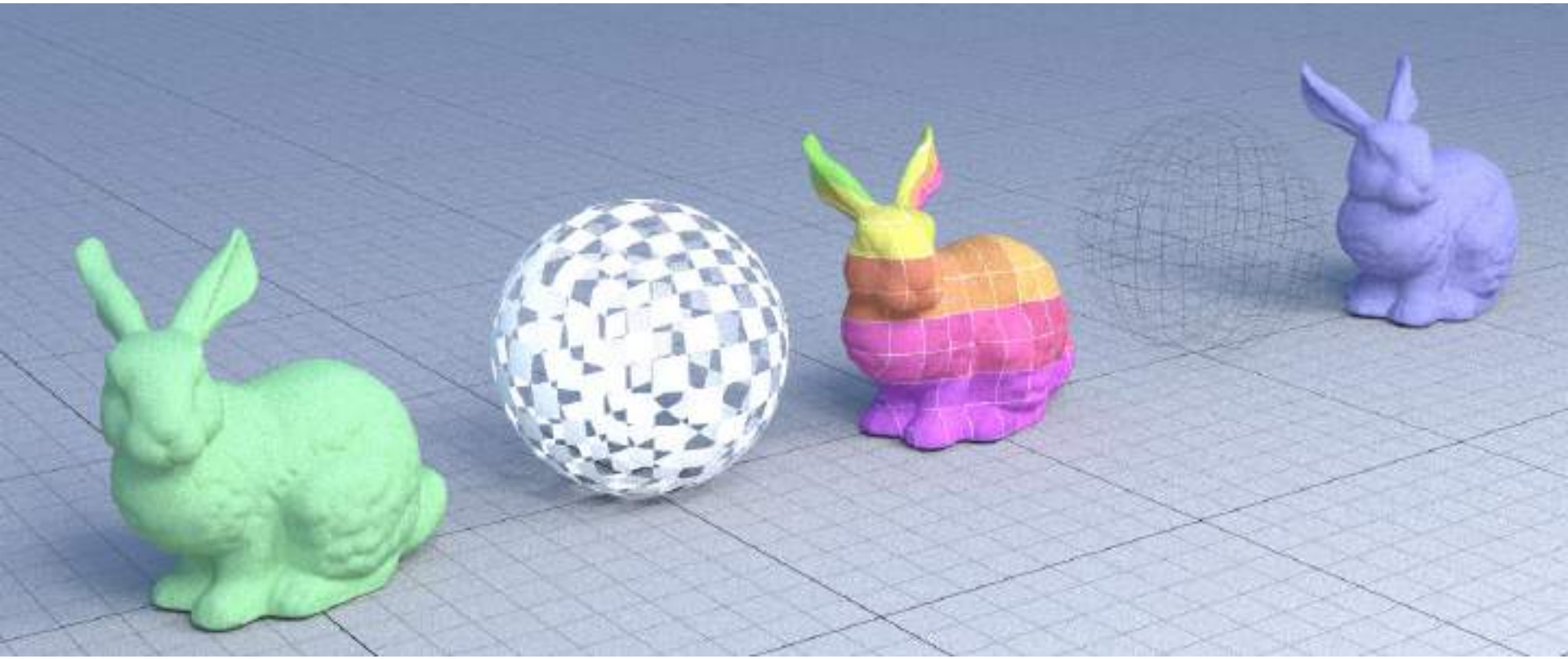
Opacity

- As we saw for images, shapes can have an opacity stored in a texture
- This is helpful to model surfaces with holes without using triangles
- We will approximate this by continuing the ray is a random number is above the surface opacity

```
<handle opacity> {
  if(rand1f(rng) > opacity)
    return shade_indirect(scene, ray3f{position, outgoing},
          bounce+1);
}
```


© 2020 Fabio Pellacini

Transparency



© 2020 Fabio Pellacini

Final Images

© 2020 Fabio Pellacini

Final Images

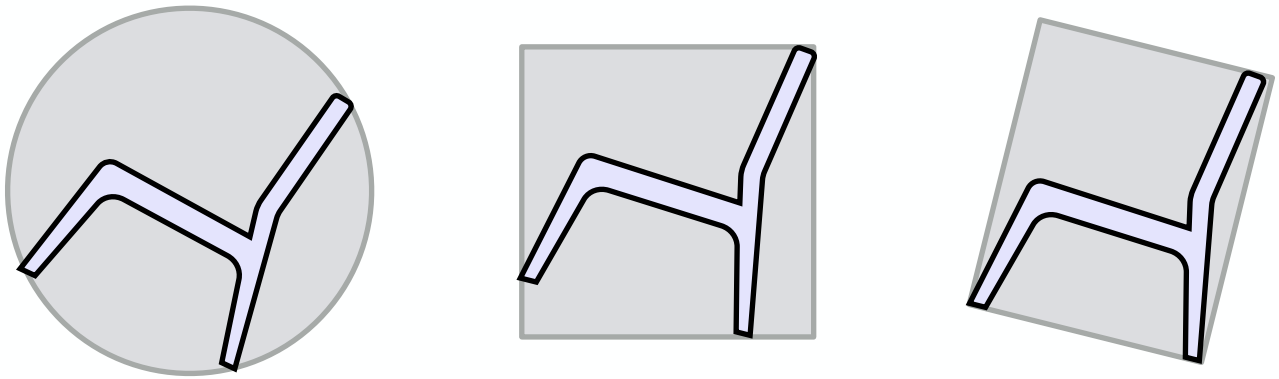
© 2020 Fabio Pellacini

Acceleration Intersections

© 2020 Fabio Pellacini

Bounding volumes

- If the object is fully contained in a volume, then if the ray doesn't hit the volume, then it doesn't hit the object
- Quick way to avoid intersections: bound object with a simple volume, then test the bounding volume (BV) first, then test object if necessary



© 2020 Fabio Pellacini

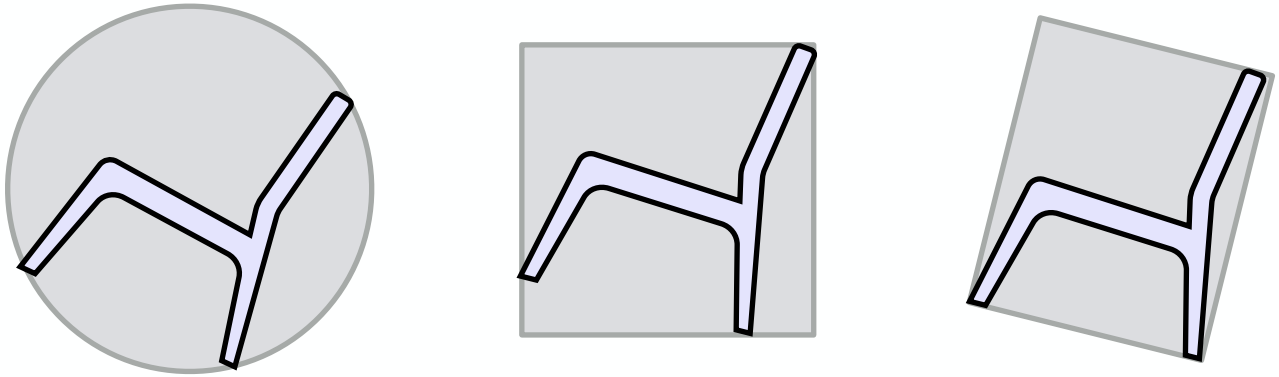
Bounding Volumes

- Cost: more for hits and near misses, less for far misses
- Cost of BV intersection test should be small
  - so use simple shapes (spheres, boxes, …)
- Cost of object intersect test should be large
  - so BV most useful for complex objects
- Tightness of fit should be good
  - loose fit leads to extra object intersections
  - tradeoff between tightness and BV intersection cost

© 2020 Fabio Pellacini

Choice of bounding volumes

- Spheres: easy to intersect, not always so tight
  - hard to handle transformed meshes
- Axis-aligned bounding boxes (AABBs): easy to intersect, often tighter, esp. for axis-aligned models
- Oriented bounding boxes (OBBs): easy to intersect with some cost for transformation, tighter for arbitrary objects
  - not easy to compute a tight for for meshes

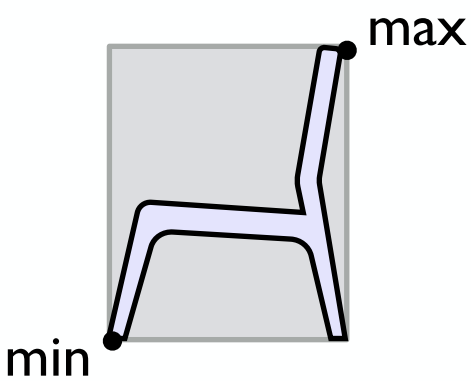


© 2020 Fabio Pellacini

Axis aligned bounding boxes

- Best tradeoff between simplicity and efficiency
- Store them by saving the minimum and maximum value in each coordinate

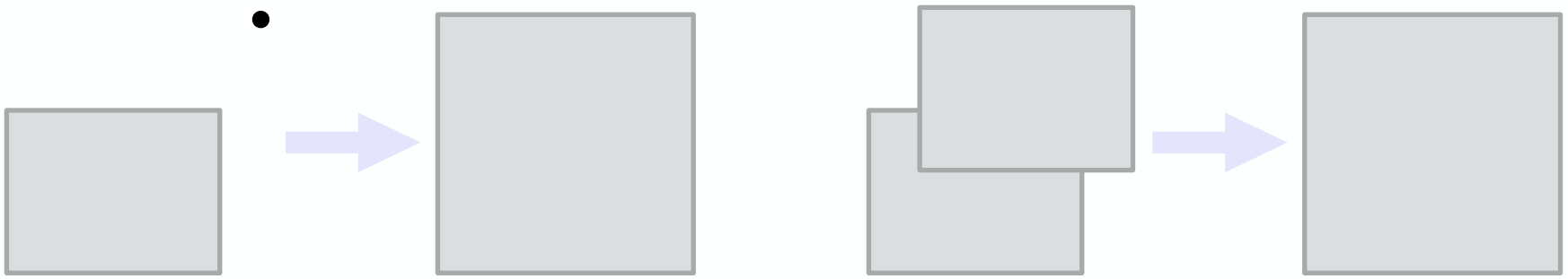
```
struct bbox3f {
  vec3f min, max;
};
```



© 2020 Fabio Pellacini

Axis aligned bounding boxes

- Helpful operations: expand with either point or other box
-



```
bbox3f merge(bbox3f a, vec3f b) {
  return { min(a.min, b), max(a.max, b) };
}

bbox3f merge(bbox3f a, bbox3f b) {
  return { min(a.min, b.min), max(a.max, b.max) };
}
```


© 2020 Fabio Pellacini

Construct AABBs

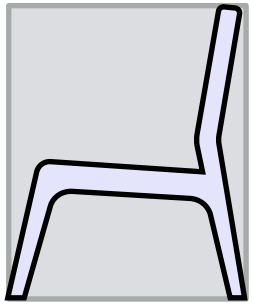
- For triangles just take the min and max of its vertices
- For parts of meshes: min and max of vertices

```
bbox3f triangle_bound(vec3f v0, vec3f v1, vec3f v2) {
  auto box = invalidb3f; // min==flt_max, max==flt_min
  box = expand_bbox(v0);
  box = expand_bbox(v1);
  box = expand_bbox(v2);
  return box;
};
```

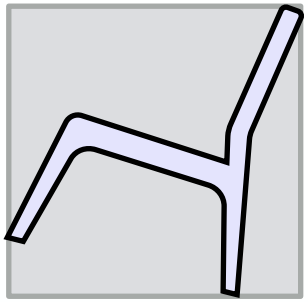

© 2020 Fabio Pellacini

Construct AABBs

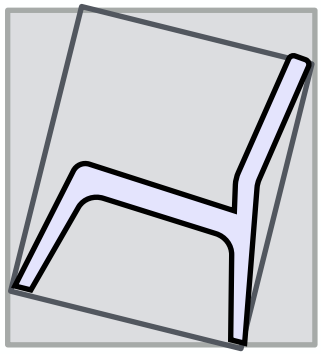
- AABBs for a surface instance
  - Transform all points for tight bound — but very inefficient
  - Most commonly take the AABB of the 8 corners of the AABB of the untransformed surface



untransformed



bounding transformed vertices

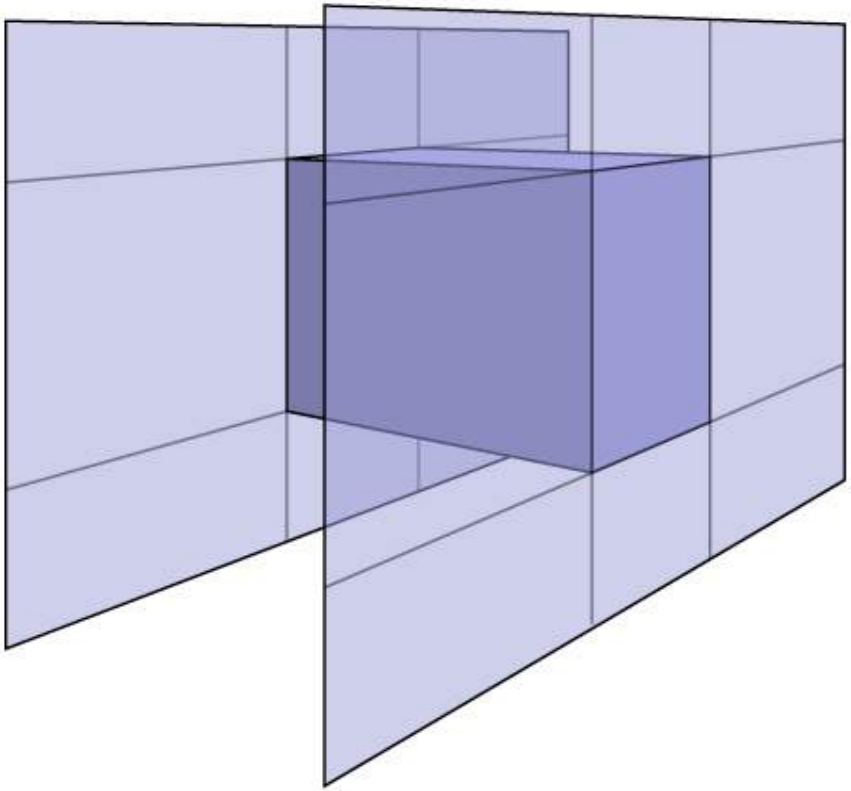


bounding transformed corners

© 2020 Fabio Pellacini

Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



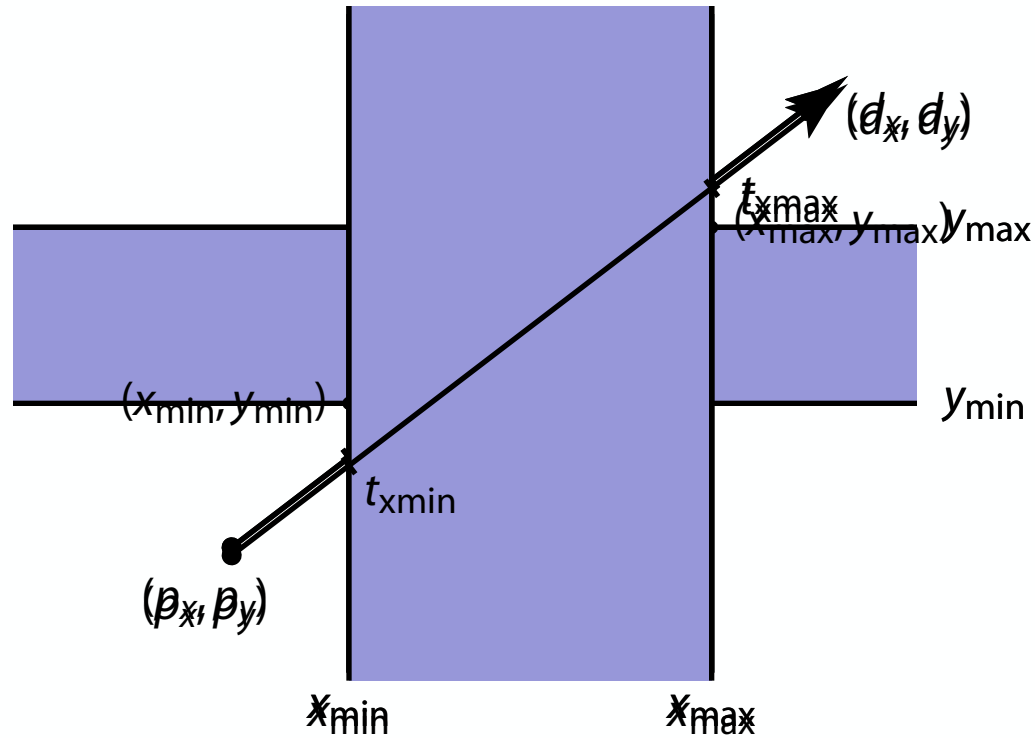
© 2020 Fabio Pellacini

Ray-slab intersection

- Show in 2D since 3D is the same

$$p_x + t_{x\min}\, d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$

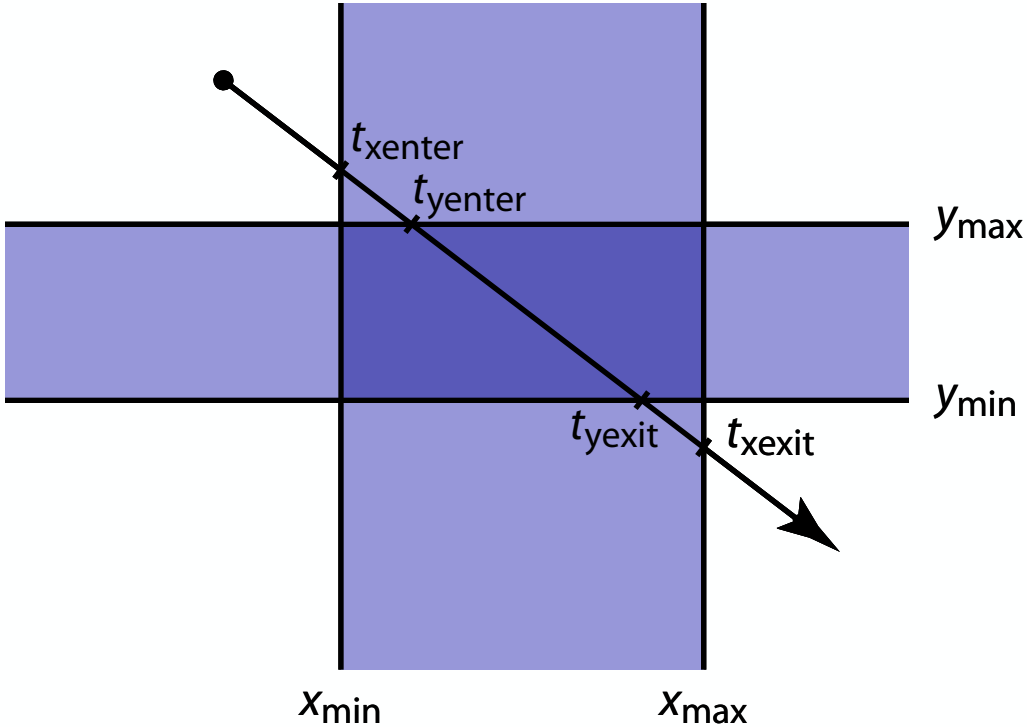


© 2020 Fabio Pellacini

Ray-box intersection

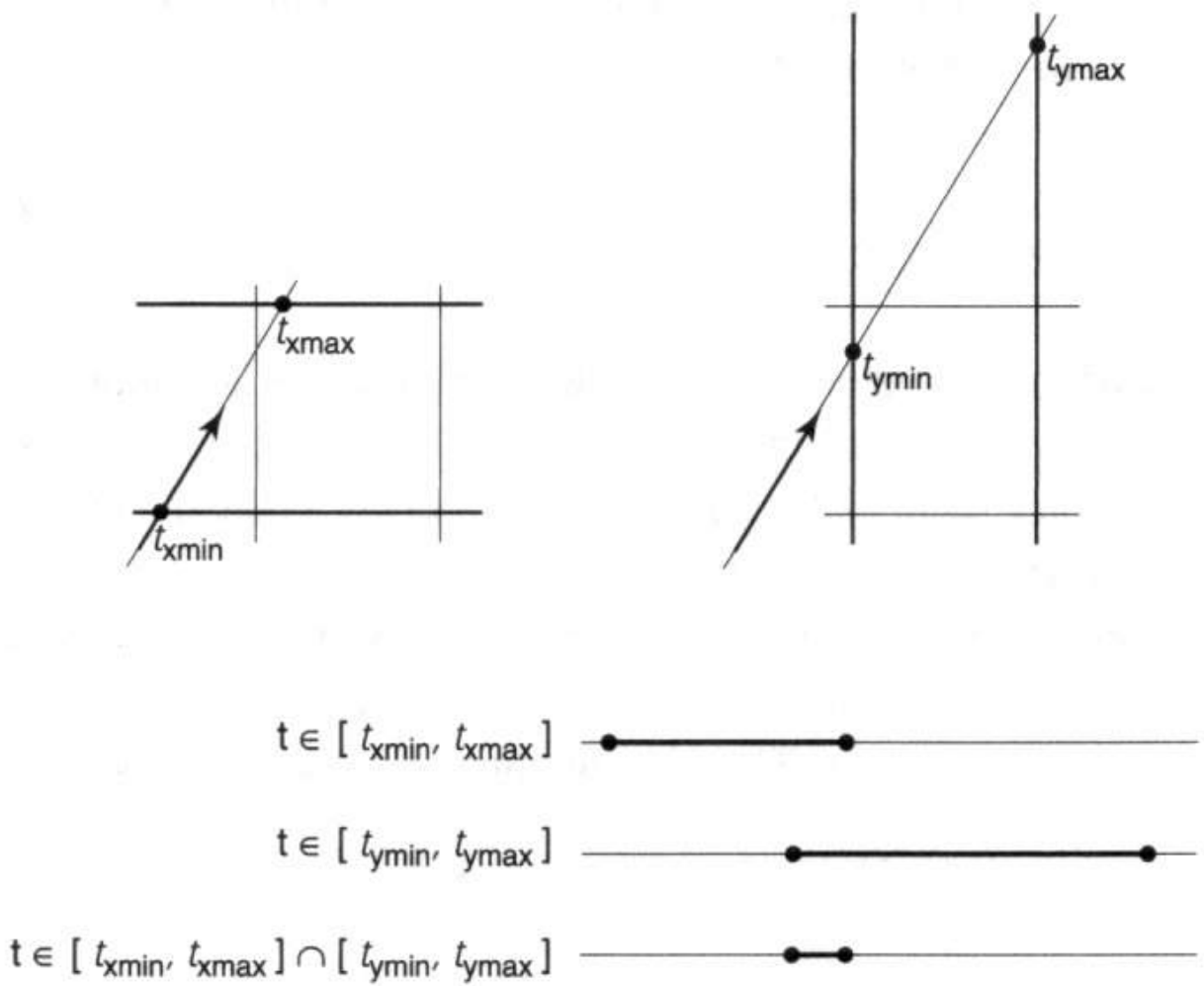
- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\text{min}}, t_{x\text{max}})$$
$$t_{x\text{exit}} = \max(t_{x\text{min}}, t_{x\text{max}})$$
$$t_{y\text{enter}} = \min(t_{y\text{min}}, t_{y\text{max}})$$
$$t_{y\text{exit}} = \max(t_{y\text{min}}, t_{y\text{max}})$$
$$t_{\text{enter}} = \max(t_{x\text{enter}}, t_{y\text{enter}})$$
$$t_{\text{exit}} = \min(t_{x\text{exit}}, t_{y\text{exit}})$$



© 2020 Fabio Pellacini

Ray-box intersection



© 2020 Fabio Pellacini

Ray-box intersection

```
// implementation 1
bool intersect_bbox(ray3f ray, bbox3f bbox) {
  auto invd = vec3f{1/ray.d.x,1/ray.d.y,1/ray.d.z};
  auto t0 = (bbox.min - ray.o) * invd;
  auto t1 = (bbox.max - ray.o) * invd;
  if (invd.x < 0) std::swap(t0.x, t1.x);
  if (invd.y < 0) std::swap(t0.y, t1.y);
  if (invd.z < 0) std::swap(t0.z, t1.z);
  auto tmin = max(t0.z, max(t0.y, max(t0.x, ray.tmin)));
  auto tmax = min(t1.z, min(t1.y, min(t1.x, ray.tmax)));
  tmax *= 1.00000024f; // numerical precision fix
  return tmin <= tmax;
}
```


© 2020 Fabio Pellacini

Ray-box intersection

```
// implementation 2
bool intersect_bbox(ray3f ray, vec3f rayd_inv, bbox3f bbox) {
  auto hit_min = (bbox.min - ray.o) * ray_dinv;
  auto hit_max = (bbox.max - ray.o) * ray_dinv;
  auto tmin   = min(hit_min, hit_max);
  auto tmax   = max(hit_min, hit_max);
  auto t0     = max(max(tmin), ray.tmin);
  auto t1     = min(min(tmax), ray.tmax);
  t1 *= 1.00000024f;
  return t0 <= t1;
}
```

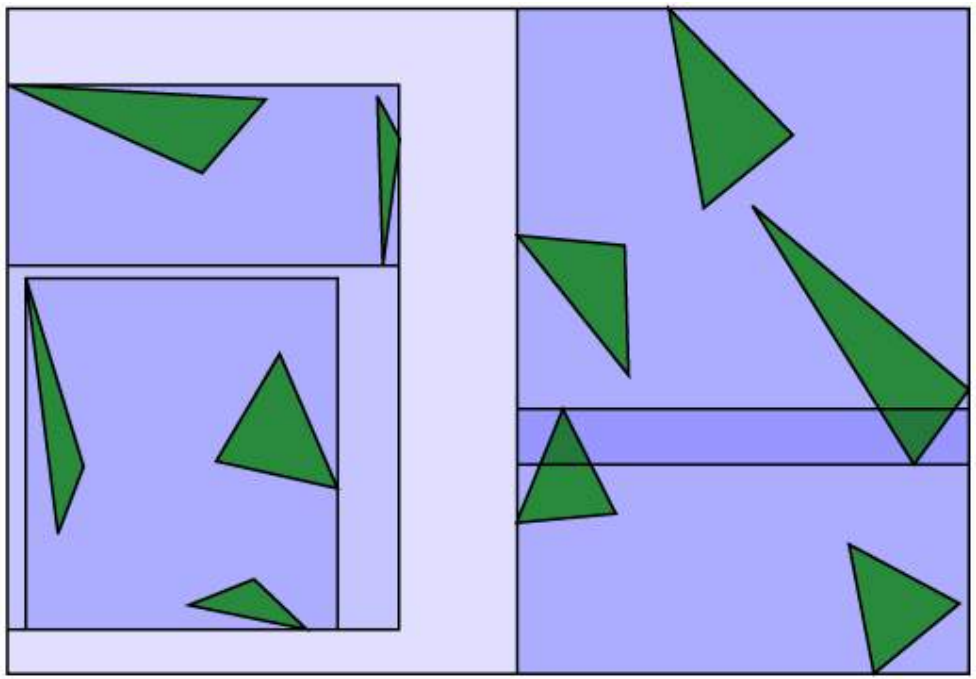

© 2020 Fabio Pellacini

Bounding Volume Hierarchy

- Bounding box around a mesh will help
- Bounding box around groups of triangles will help
- Bounding box around groups of meshes will help
- Leads to the idea of using bounding volumes all the way from the whole scene down to groups of a few objects
- BVH: bounding volume hierarchy

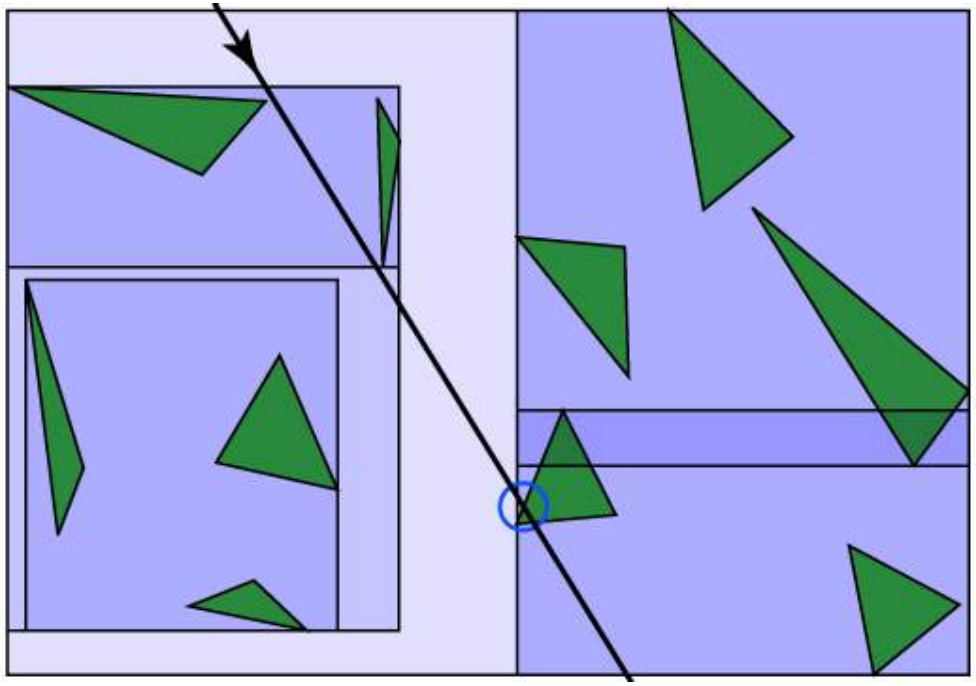
© 2020 Fabio Pellacini

BVH construction example



© 2020 Fabio Pellacini

BVH ray-tracing example



© 2020 Fabio Pellacini

Implementing BVHs

- For efficiency, avoid trees with pointers
- Use a flattened, or linear, BVH where all nodes are stored in one array and all primitives in another
- Internal nodes have indices to the node array
- Leaf nodes have indices to the leaves

```
struct bvh_node {
  bbox3f bbox;
  uint32 start;
  uint16 num;
  uint16 internal;
};
```

```
struct bvh_tree {
  vector<bvh_node> nodes;
  vector<int> primitives;
};
```


© 2020 Fabio Pellacini

```
bool intersect_bvh_slow(bvh_tree* bvh, int node_id,
  shape* shape, ray3f ray, intersection* isec) {
  auto node = bvh->nodes[node_id];
  if (!intersect_bbox(ray, node.bbox)) return false;
  auto hit = false;
  if (node.internal) {
   for (auto i = 0; i < node.count; i++) {
      if(!intersect_bvh_slow(bvh, node.start+i, shape, ray,
           isec) continue;
      hit = true;
    }
 } else {
   for (auto i = 0; i < node.count; i++) {
      auto t = shape->triangles[bvh->primitives[i]];
      if (!intersect_triangle(ray, shape->positions[t.x],
          shape->positions[t.y], shape->positions[t.z],
          isec->distance, isec->uv)) continue;
      hit = true; ray.tmax = dist;
      isec->element = bvh->primitives[i];
 }
 return hit;
}
```


© 2020 Fabio Pellacini

```
bool intersect_bvh(bvh_tree* bvh, shape* shape, ray3f ray,
                   intersection* isec) {
  int node_stack[64]; auto node_cur = 0;
  node_stack[node_cur++] = 0;
  auto hit = false;
  while (node_cur) {
    auto node = bvh->nodes[node_stack[--node_cur]];
    if (!intersect_bbox(ray, node.bbox)) continue;
    if (node.internal) {
       for (auto i = 0; i < node.count; i++)
         node_stack[node_cur++] = i + node.start;
    } else {
      for (auto i = 0; i < node.count; i++) {
        auto t = shape->triangles[bvh->primitives[i]];
        if (!intersect_triangle(ray, shape->positions[t.x],
            shape->positions[t.y], shape->positions[t.z],
            isec->distance, isec->uv)) continue;
        hit = true; ray.tmax = dist;
        isec->element = bvh->primitives[i];
      }
    }
  }
  return hit;
}
```


© 2020 Fabio Pellacini

Building a hierarchy

- Usually do it top-down
- Make box for whole scene, then split into two parts
- Recurse on parts, stopping when there are too few objects
- Split by sorting along longest axis
- Many heuristics
  - split in the middle of the parent box — split space, but not number of primitives
  - split into two equal number of components — split primitives, but may leads to overlapping boxes
  - surface area heuristic: split for best ray cost — hard to implement but leads to better intersection

© 2020 Fabio Pellacini

```
// generic primitive
struct bvh_prim {
  bbox3f bbox;   // bounding box
  vec3f center;  // bounding box center
  int primitive; // primitive id
};

void build_bvh(bvh_tree* bvh, shape* shape, bool equal_num) {
  auto prims = std::vector<bvh_prim>();
  for (auto ei = 0; ei < shape->triangles.size(); ei++) {
    auto e = shape->triangles[ei];
    auto bbox = triangle_bound(shape->positions[e.x],
      shape->positions[e.y], shape->positions[e.z]);
    prims.push_back({bbox, (bbox.min + bbox.max) / 2, ei});
  }
  build_bvh(bvh, bound_prims, equal_num);
}
```


© 2020 Fabio Pellacini

```
// recursively build a BVH by partitioning and sorting
// the prim array
void build_bvh(bvh_tree* bvh, vector<bvh_prim>& bound_prims,
    bool equal_num) {
  // prepare the tree and reserve memory
  bvh->nodes.reserve(prims.size() * 2);
  bvh->nodes.push_back({});
  // recursively partition and sort the prim array
  make_node(bvh, 0, prims, 0, prims.size(), equal_num);
  bvh->nodes.shrink_to_fit();
  // copy the leaf nodes back
  bvh->leaf_prims.resize(prims.size());
  for (int i = 0; i < prims.size(); i++) {
    bvh->leaf_prims[i] = bound_prims[i].pid;
  }
  // done
  return bvh;
}
```


© 2020 Fabio Pellacini

```
void make_node(bvh_tree* bvh, int node_id,
    vector<bound_prim>&  prims, int start, int end) {
    auto node = &bvh->nodes[node_id];            // grab node
    node->bbox = invalidb3f;                     // compute bbox
    for (auto i = start; i < end; i++)
        node->bbox = merge(node->bbox, prims[i].bbox);
    auto split = false; auto axis = 0; auto mid = (start+end)/2;
    if (end - start > 4)        // split prims and get mid point
        split = split_prims(prims,start,end,&axis,&mid);
    if(!split) {                                  // make a leaf
        node->internal = false;
        node->start = start;
        node->count = end-start;
    } else {                              // make an internal node
        node->internal = true;
        node->axis = axis;
        auto first = (int)bvh->nodes.size();
        node->start = first;
        node->count = 2;
        // allocate children and recurse
        bvh->nodes.push_back({}); bvh->nodes.push_back({});
        make_node(bvh, first, prims, start, mid);
        make_node(bvh, first + 1, prims, mid, end);
    }
}
```


© 2020 Fabio Pellacini

```
bool split_prims_equalnum(vector<bound_prim>& prims,
    int start, int end, int* axis, int* mid) {
  // pick axis by choosing the largest axis of the box
  // use centers to avoid issues with large primitives
  auto cbbox = invalidb3f;
  for (auto i = start; i < end; i++)
    cbbox = merge(cbbox, prims[i].center);
  auto size = cbbox.max - cbbox.min;
  if (size == vec3f{0, 0, 0}) return false;
  // pick axis and mid
  *axis = max_index(size); // pick largest axis
  *mid = (start + end) / 2; // split elements in half
  // arrange the array to have elements divided in two groups
  // w.r.t the middle element; smaller on the left, and larger
  // on the right; C++ std::nth_element does that
  std::nth_element(prims.begin() + start,
                   prims.begin() + *mid,
                   prims.begin() + end,
                   [axis](auto a, auto b) {
                     return a.center[*axis] < b.center[*axis];
                   });
  return true;
}
```

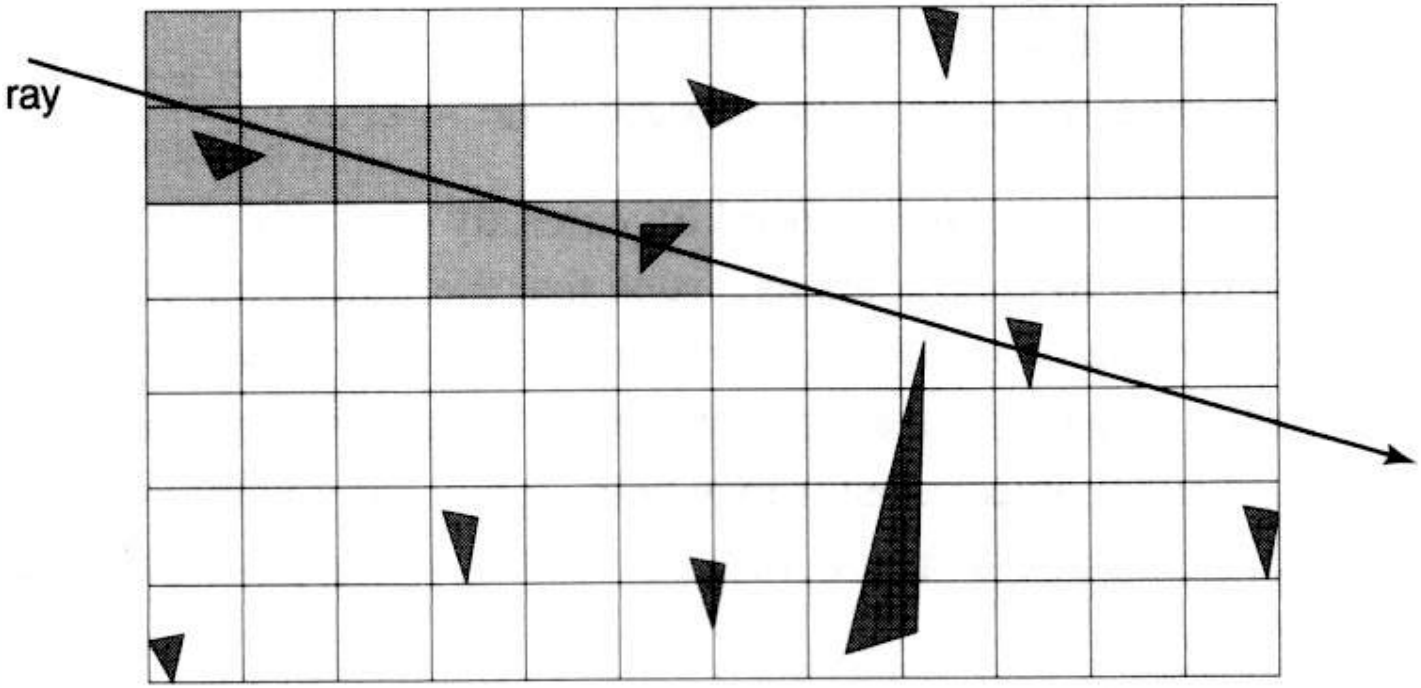

© 2020 Fabio Pellacini

```
bool split_prims_equalsize(vector<bound_prim>& prims,
    int start, int end, int* axis, int* mid) {
  // pick axis by choosing the largest axis of the box
  // use centers to avoid issues with large primitives
  auto cbbox = invalidb3f;
  for (auto i = start; i < end; i++)
    cbbox = merge(cbbox, prim[i].center);
  auto csize = cbbox.max - cbbox.min;
  if (csize == vec3f{0, 0, 0}) return false;
  // pick axis and split center
  *axis = max_index(csize); // pick largest axis
  // split space in half by the half plane of the bbox
  auto half = (cbbox.min + cbbox.max) / 2;
  // partition the array such that all left elements are
  // less or equal to half and all right elements are greater;
  // compute the split point mid; C++ std::partition does that
  *mid = (int)(std::partition(prims.begin() + start,
        prims.begin() + end,
        [axis, half](auto a) {
            return a.center[*axis] < half[*axis];
        }) - prims.begin());
  return true;
}
```


© 2020 Fabio Pellacini

Regular space subdivision

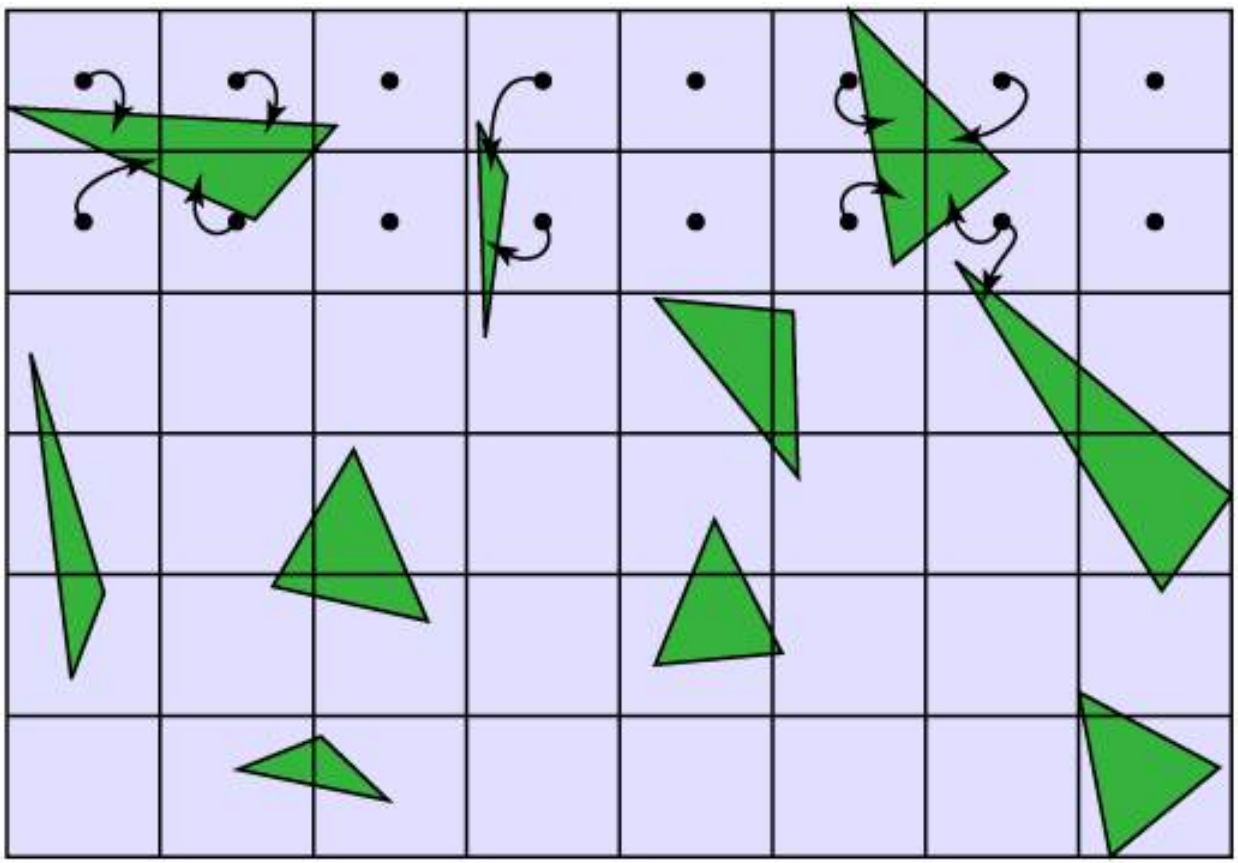
- An entirely different approach: uniform grid of cells



© 2020 Fabio Pellacini

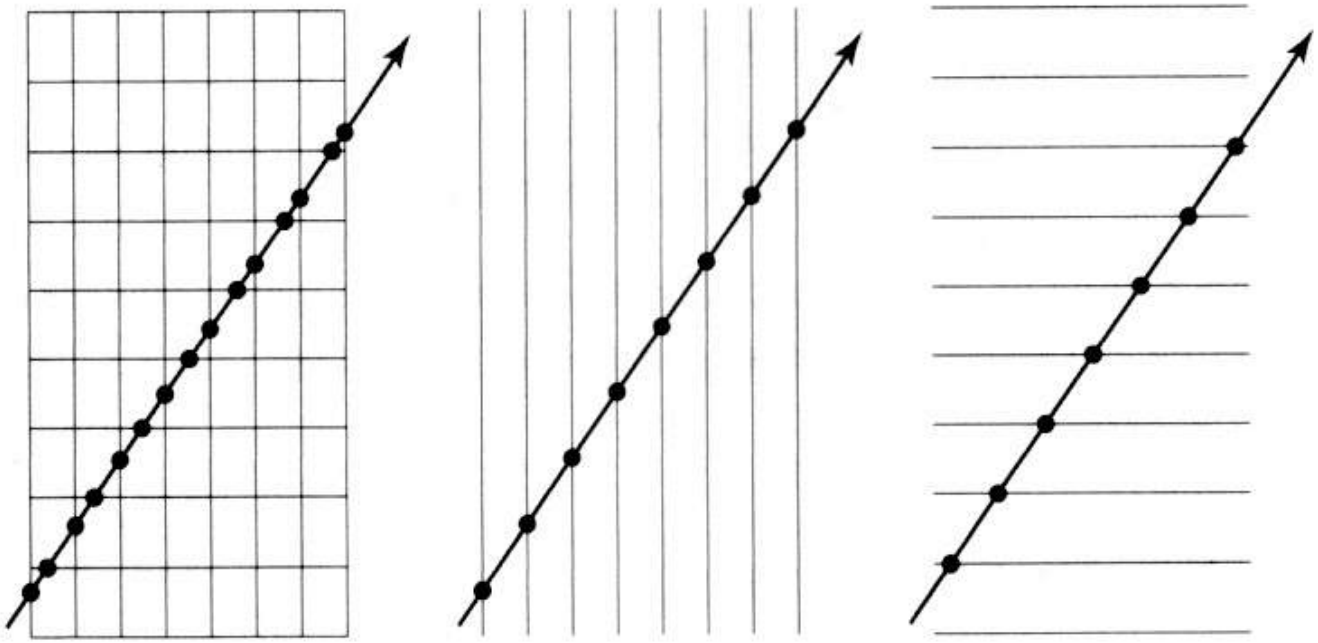
Regular grid example

- Grid divides space, not objects



© 2020 Fabio Pellacini

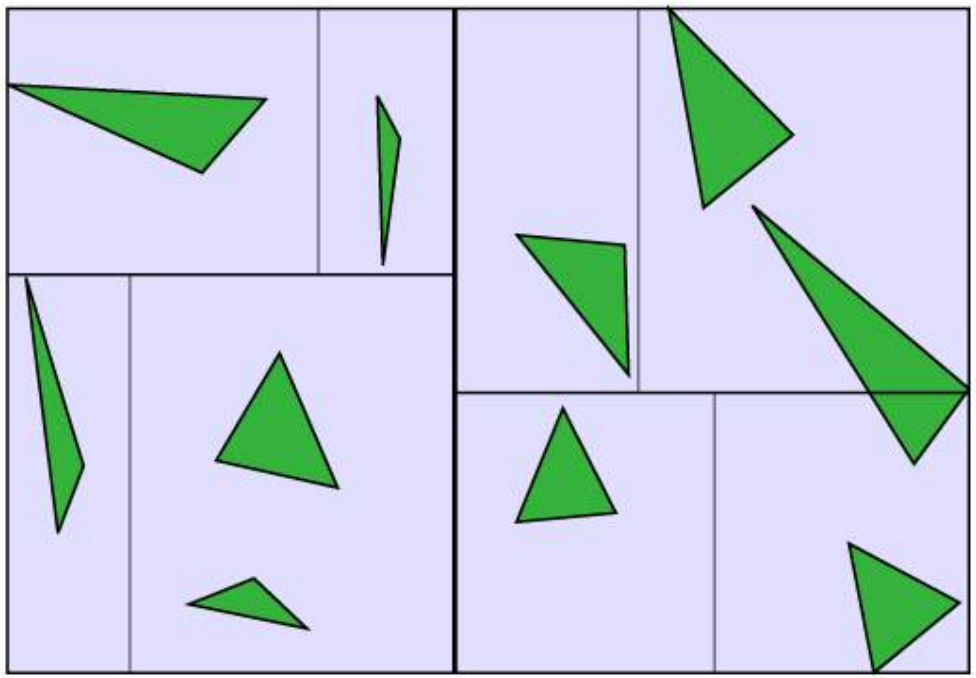
Traversing a regular grid



© 2020 Fabio Pellacini

kd-Tree subdivision

- Subdivides space, like grid, but remains adaptive, like BVH



© 2020 Fabio Pellacini

Practical Acceleration — BVH

- Numerically stable intersection
- Primitives are references only one — bounded memory usage
- Very good intersection performance
- Relatively easy to build, maybe not optimally

© 2020 Fabio Pellacini

Representing the World

Prof. Fabio Pellacini

© 2020 Fabio Pellacini •

[WikimediaCommons]

© 2020 Fabio Pellacini •

Scene elements

- To make an image, we need to represent
  - cameras
  - shapes
  - materials
- To make a movie, we need to also represent
  - animation (later)

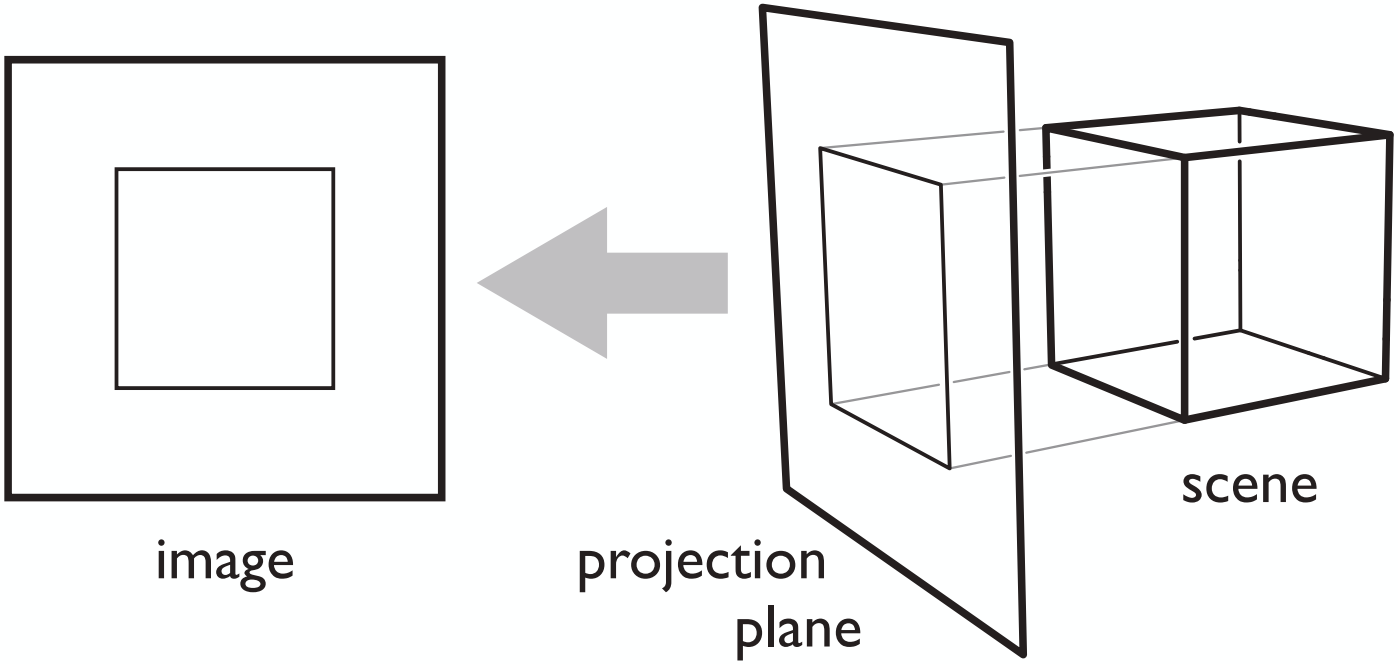
© 2020 Fabio Pellacini

Representing Cameras

© 2017 Fabio Pellacini and Steve Marschner

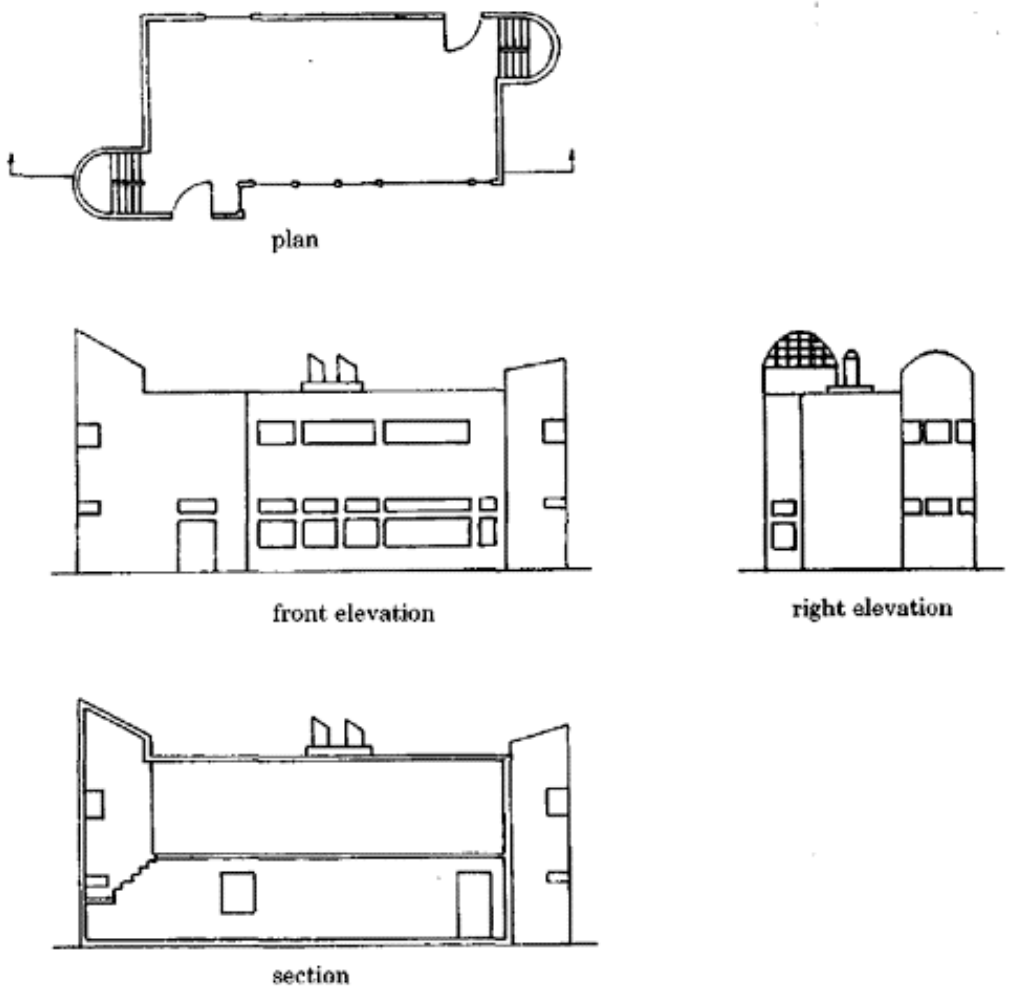
Orthographic Projections

- To render an image of a 3D scene, we project it onto a plane
- Simplest type is parallel projection
- Mostly used in modeling interfaces



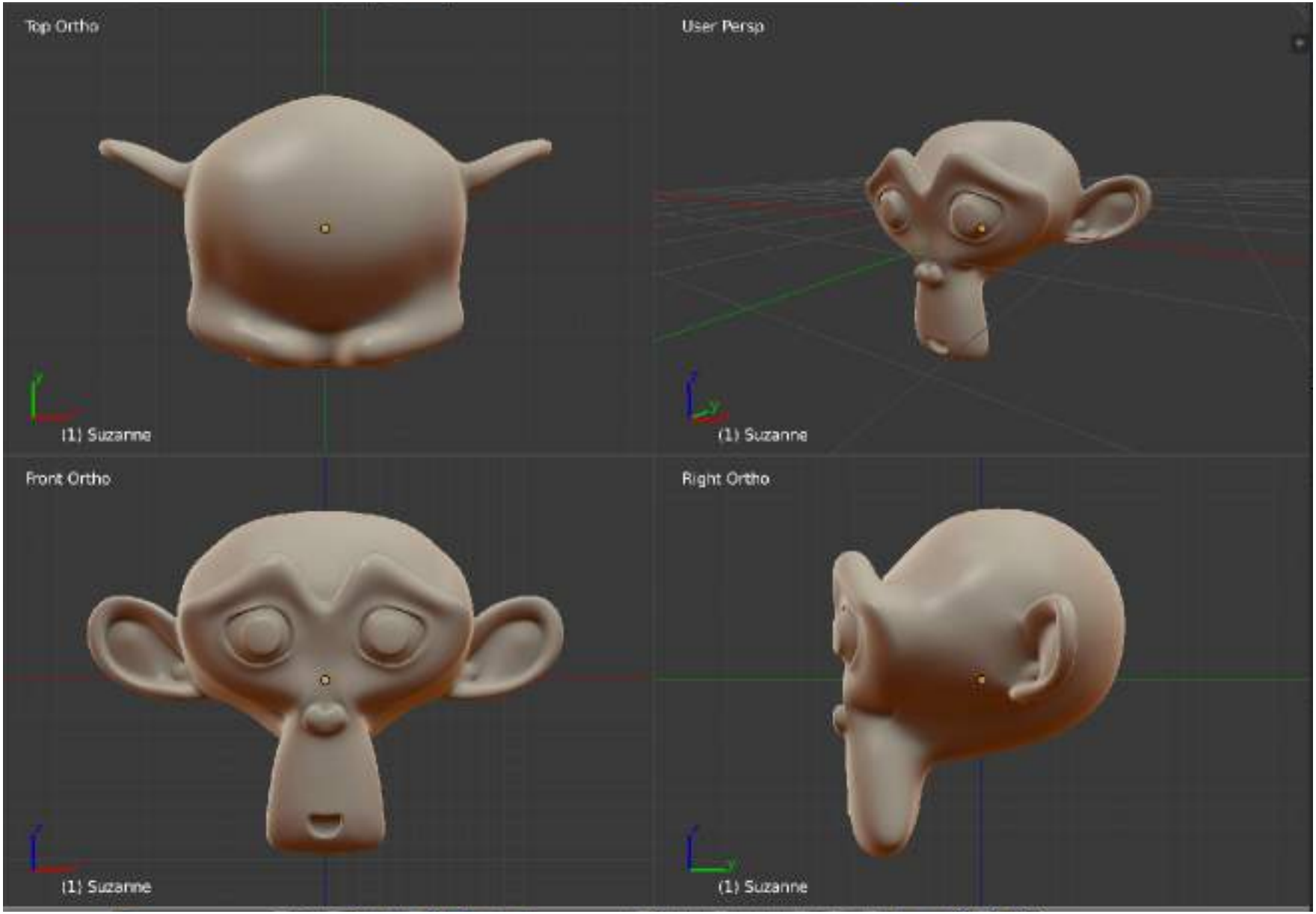
© 2020 Fabio Pellacini •

Orthographic Projection



[Carlbom & Paciorek 78]

© 2020 Fabio Pellacini •

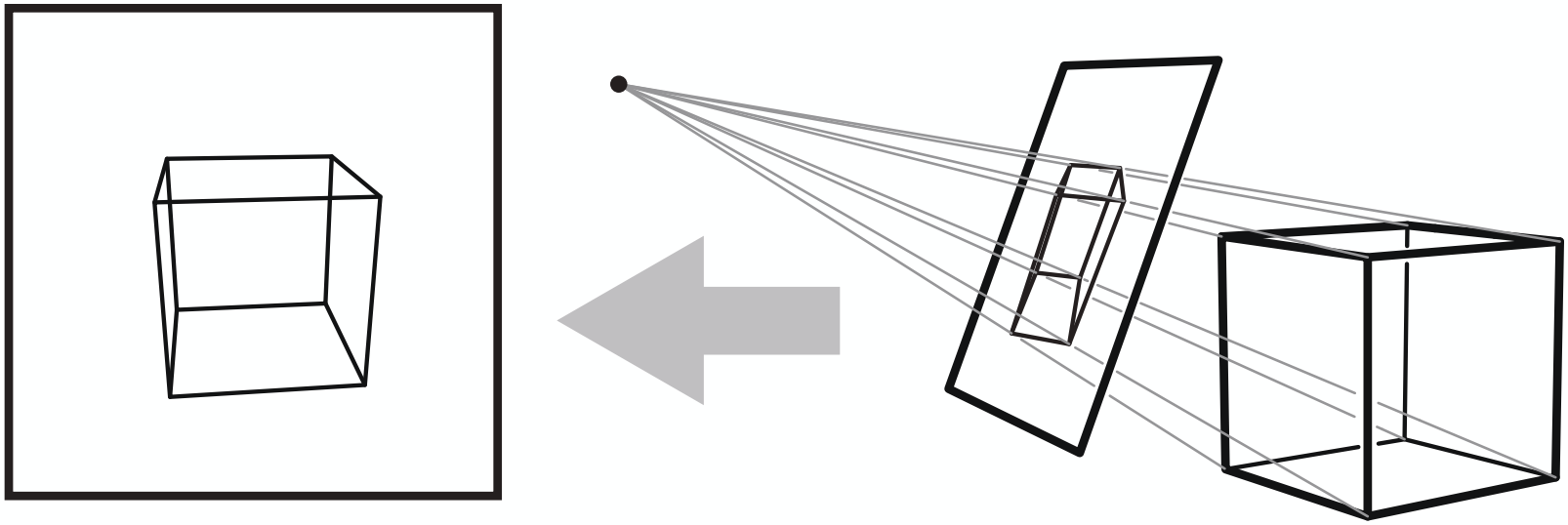


[Blender]

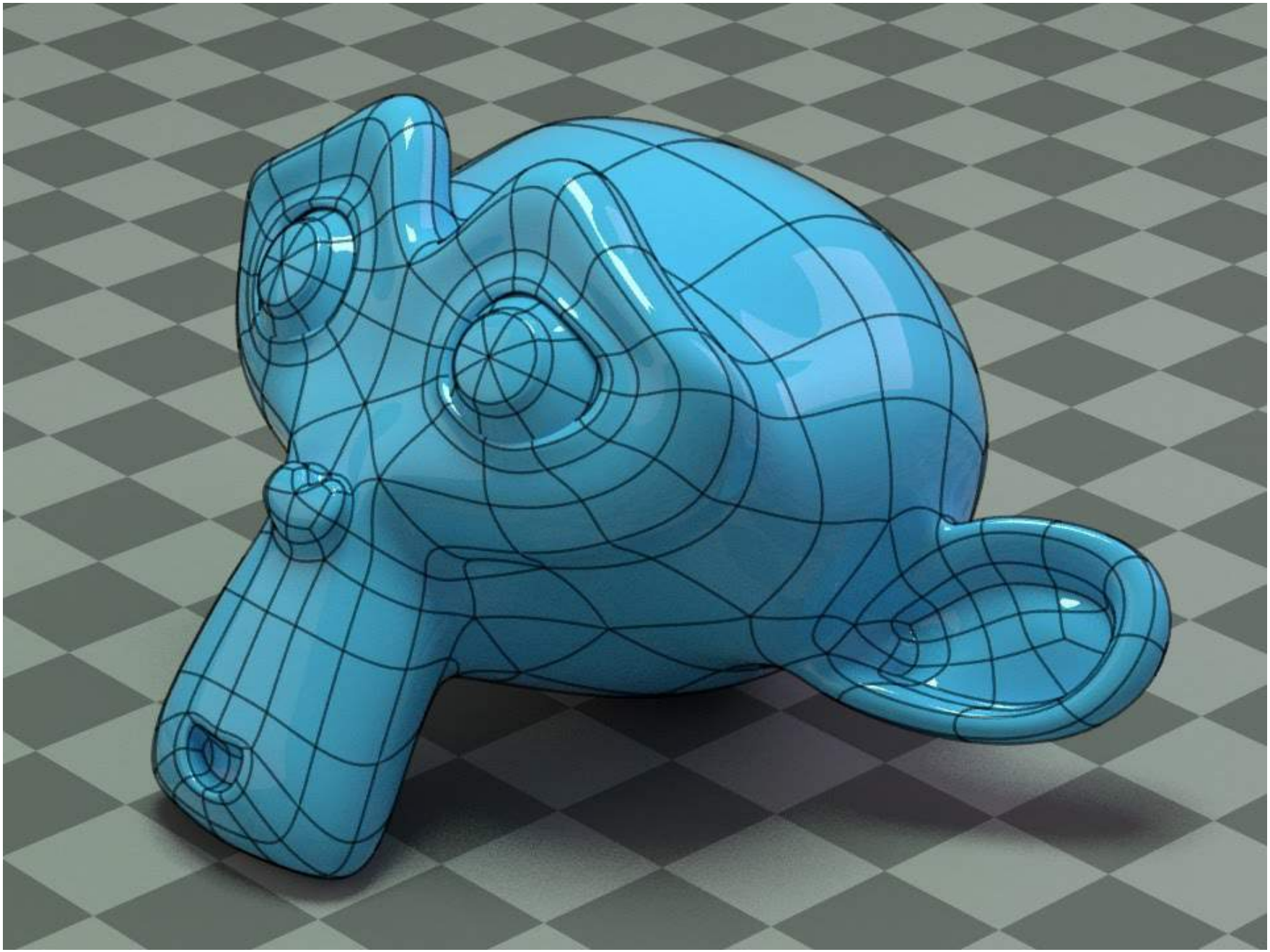
© 2020 Fabio Pellacini •

Perspective Projections

- Orthographic projection do not emulate a real camera
- Most common projection type is perspective projection
- Simulates a real pinhole camera

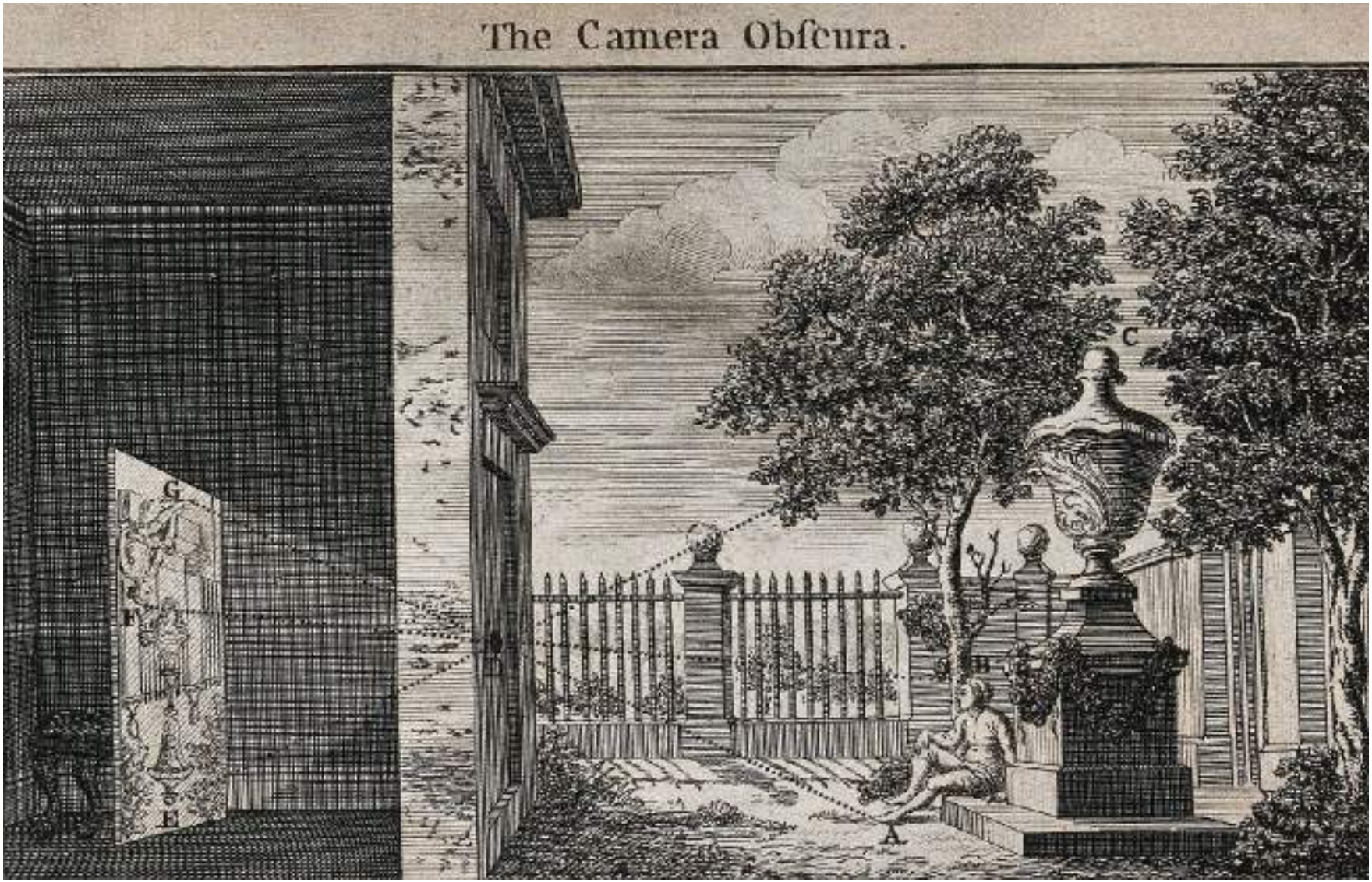


© 2020 Fabio Pellacini



[YouTueb - Blender Nerd]

© 2020 Fabio Pellacini •



[WikipediaCommons]

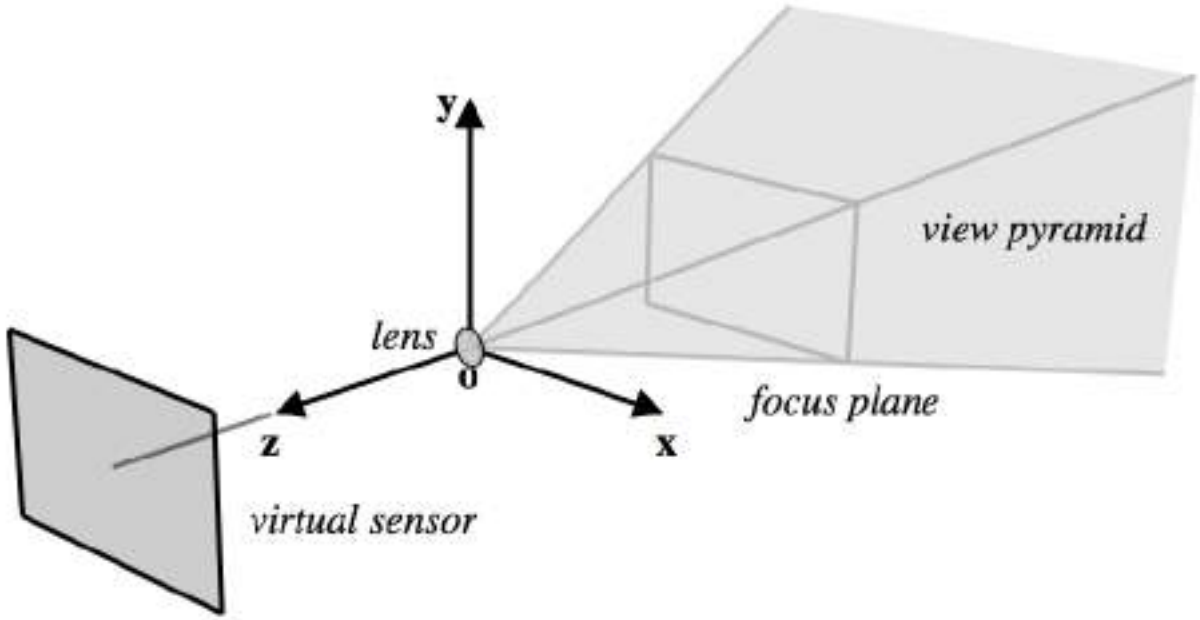
© 2020 Fabio Pellacini

[PetaPixel.com]

© 2020 Fabio Pellacini

Camera Representation

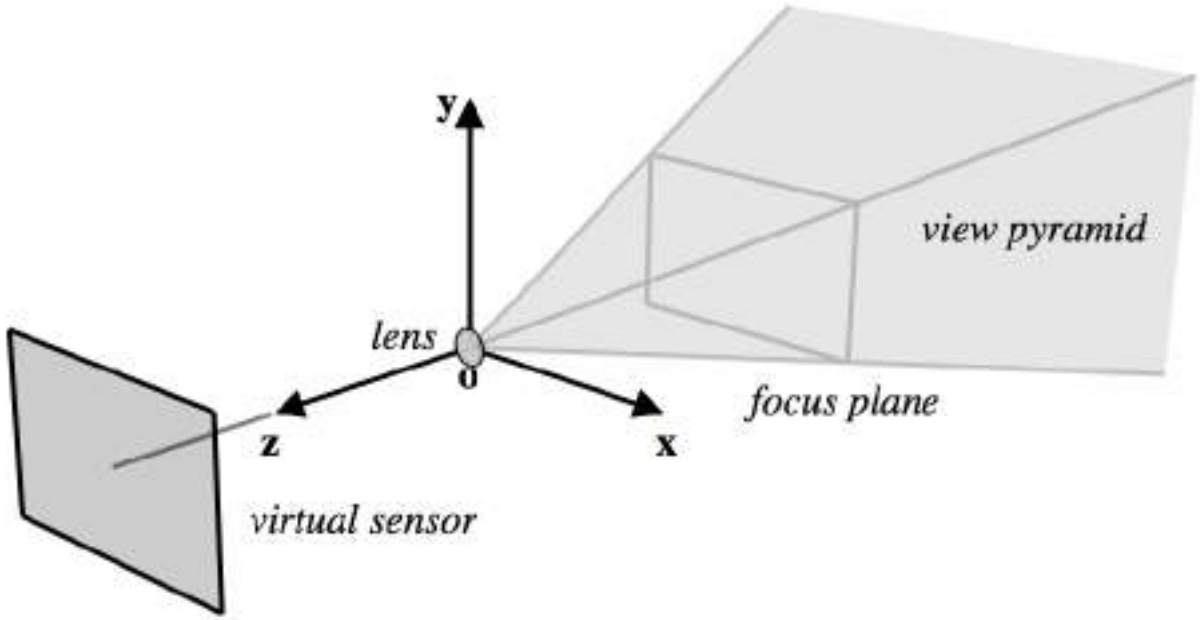
- The camera position and orientation are defined by a frame
- The camera projection is defined by its lens field of view and the distance to the camera sensor
- The camera focus is defined by the lens aperture and the focus distance



© 2020 Fabio Pellacini

Camera Representation

```
struct camera {
  frame3f frame;
  float lens, aspect, film;
  float aperture, focus;
};
```



© 2020 Fabio Pellacini

Defining Camera Frame

- Providing the frame directly is unintuitive
- Calculate frame camera position and view direction
  - camera position **o,** where the camera is, set as frame origin
  - view direction **-z**: which way the camera is looking
  - up vector **y**: how the camera is oriented, generally world up
- Since view direction and up vector are not orthogonal, use frame building code shown before

© 2020 Fabio Pellacini

Representing Shapes

© 2017 Fabio Pellacini and Steve Marschner

Different Representations

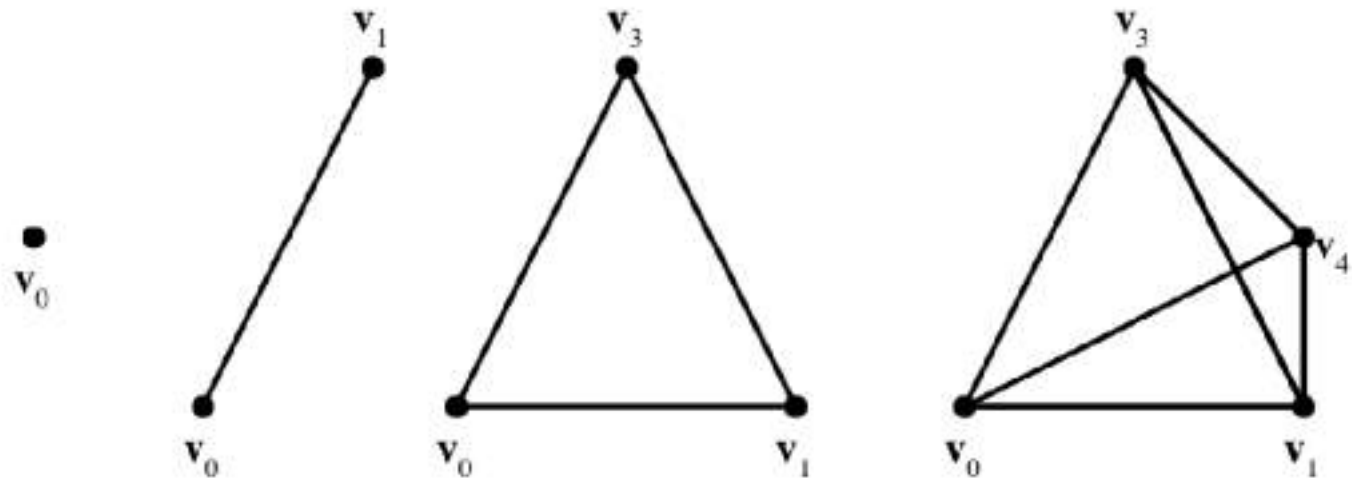
- Different representations are sometimes needed
- Optimize for different shape "types"
- Optimize for different algorithms, e.g. rendering vs simulation
- Support intuitive and efficient editing by artists

- But this implies significant software complexity
  - cannot mix-and-match algorithms
  - hard to exchange shapes
- Trend toward using a fewer representations

© 2020 Fabio Pellacini

Points, Lines and Triangles

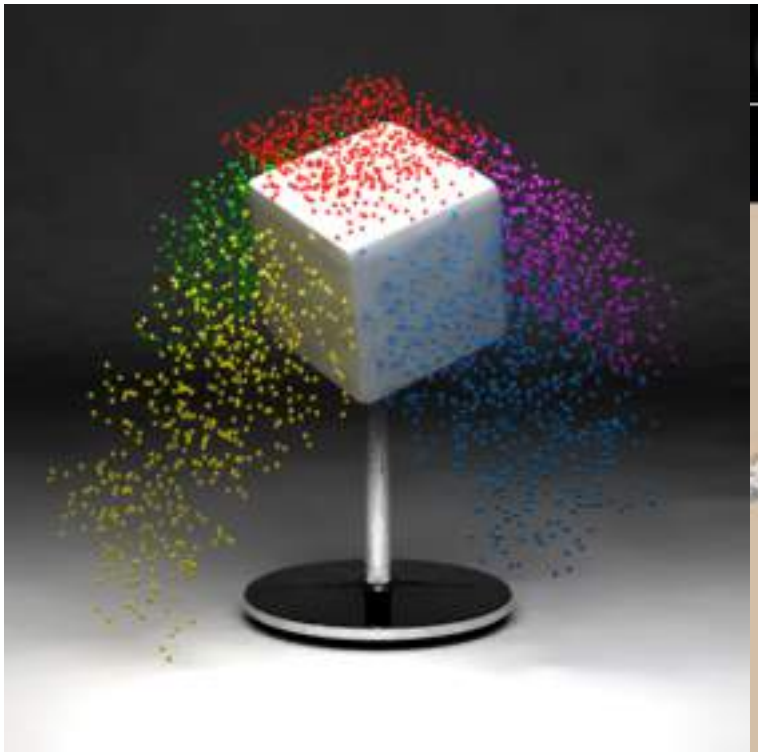
- For many algorithms, we can describe the world as collections of points, lines and triangles
- For simulation, it is sometimes helpful to use tetrahedra to represent the objects' volumes

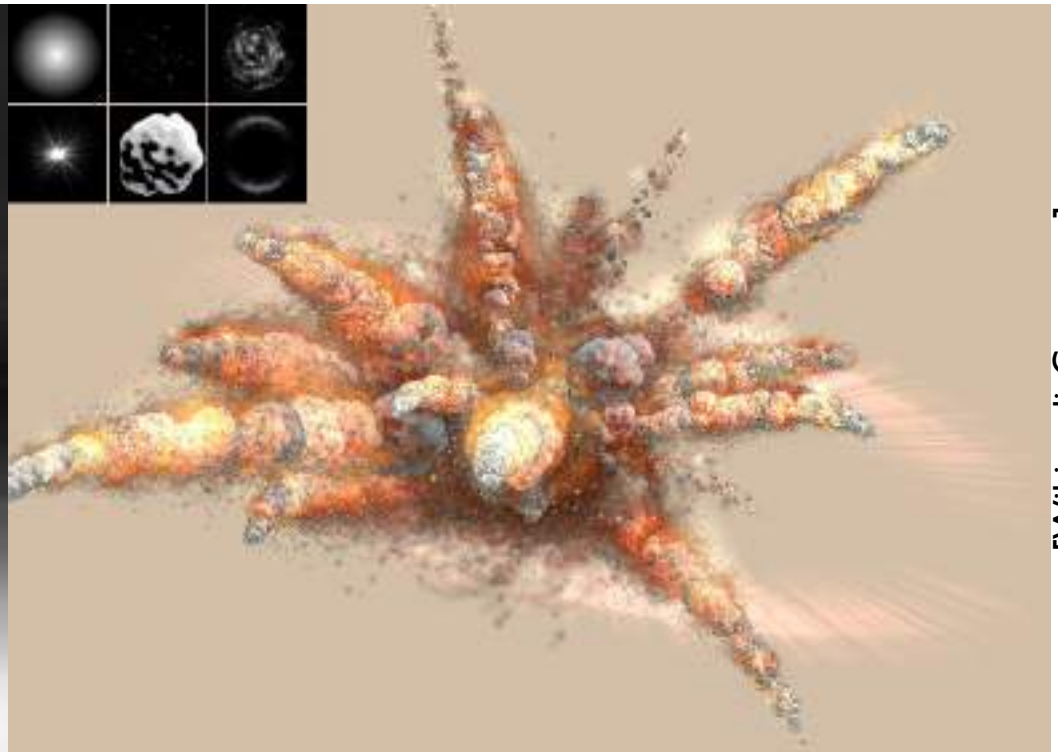


© 2020 Fabio Pellacini

Points

- Defined by one vertex
- Small enough that their properties do not vary over their "surface"
- Used for small particles (like ember, dust, etc) and simulation





[WikimediaCommons]

© 2020 Fabio Pellacini

Line segments

- Defined by two vertices
- Properties vary along the segment
- Used to represent hair and 2D outlines

[Blender]

© 2020 Fabio Pellacini

Line segments

- Line tangent and length defined as

$$\mathbf{t} = \frac{\mathbf{v}_1 - \mathbf{v}_0}{|\mathbf{v}_1 - \mathbf{v}_0|}$$
$$l = |\mathbf{v}_1 - \mathbf{v}_0|$$

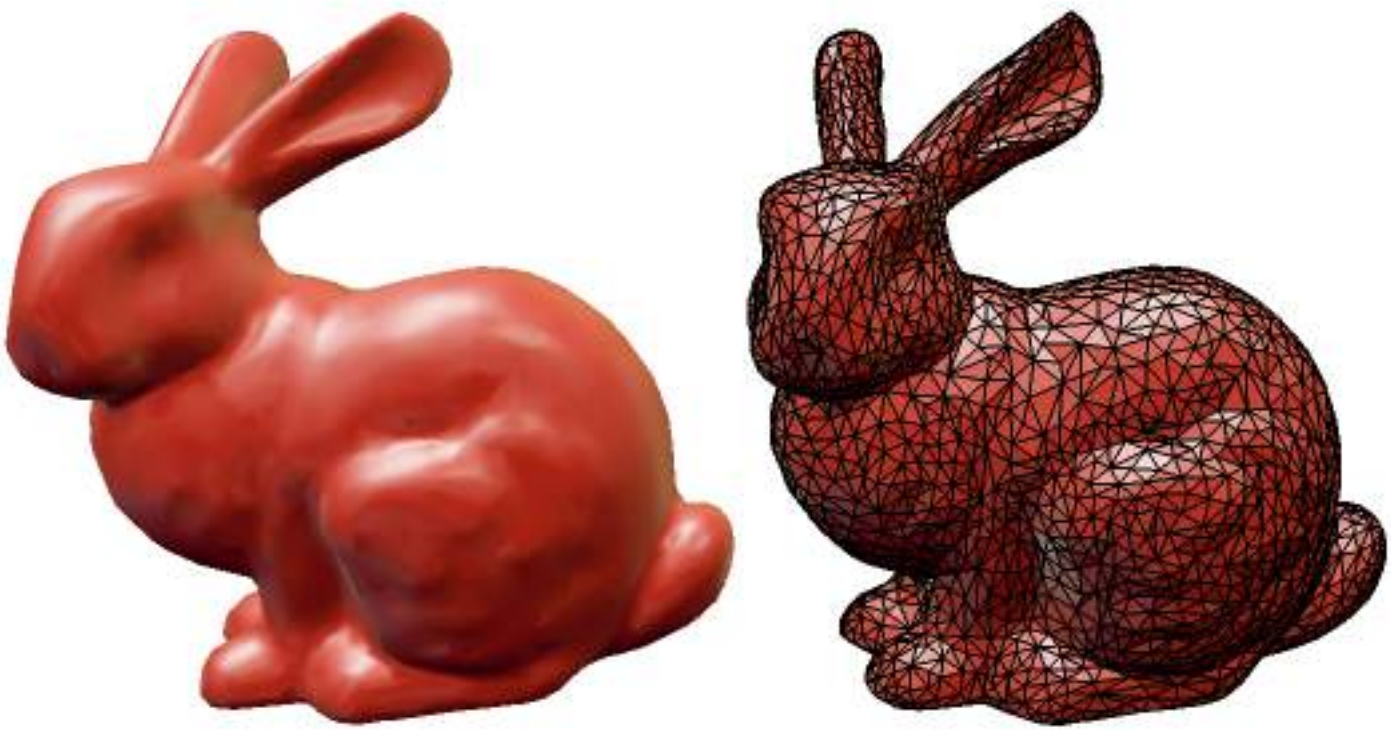
- Points on a segment can be written as linear combination between the vertices

$$\mathbf{p}(t) = (1-t)\mathbf{v}_0 + t\mathbf{v}_1 \quad t \in [0,1]$$

© 2020 Fabio Pellacini

Triangles

- Defined by three vertices
- Flat shape contained in the plane passing through those vertices
- Used to represent surfaces



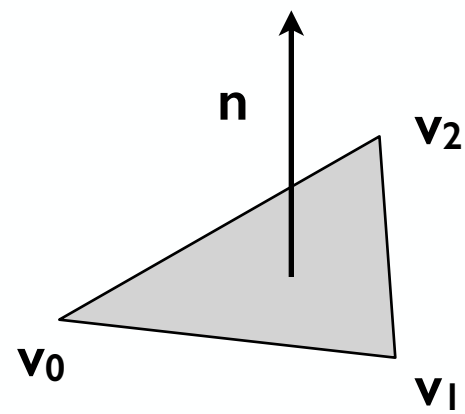
© 2020 Fabio Pellacini

Triangles

- Vector normal to the plane is the triangle's normal, area computed from cross product formulation
- Convention on orientation: vertices are counter-clockwise as seen from "front", surface normal points"outward"

$$\mathbf{n} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}$$

$$A = \frac{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}{2}$$

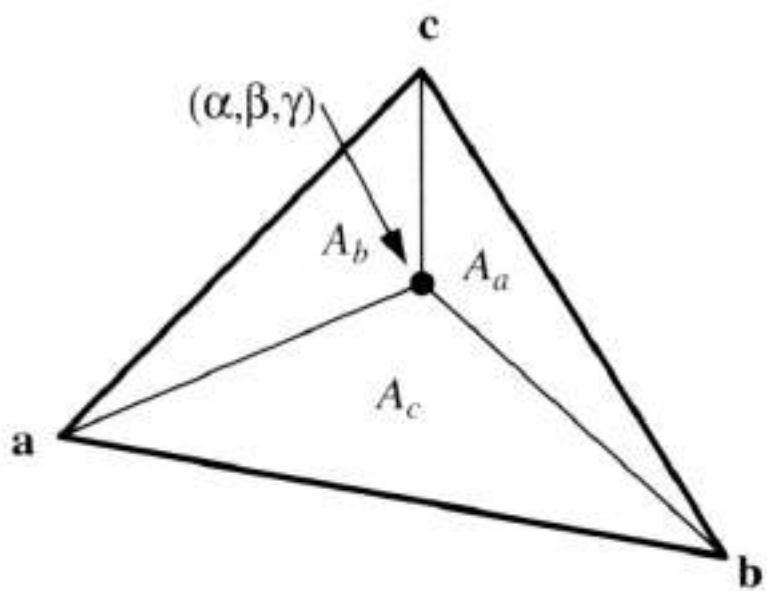


© 2020 Fabio Pellacini

Barycentric coordinates

- Points on a triangles can be written as linear combination of the vertices — the weights are called *barycentric coordinates*

$$\mathbf{p} = w_0\mathbf{v}_0 + w_1\mathbf{v}_1 + w_2\mathbf{v}_2$$
$$w_0 + w_1 + w_2 = 1$$
$$w_0 \geq 0 \quad w_1 \geq 0 \quad w_2 \geq 0$$



[Shirley 2000]

© 2020 Fabio Pellacini

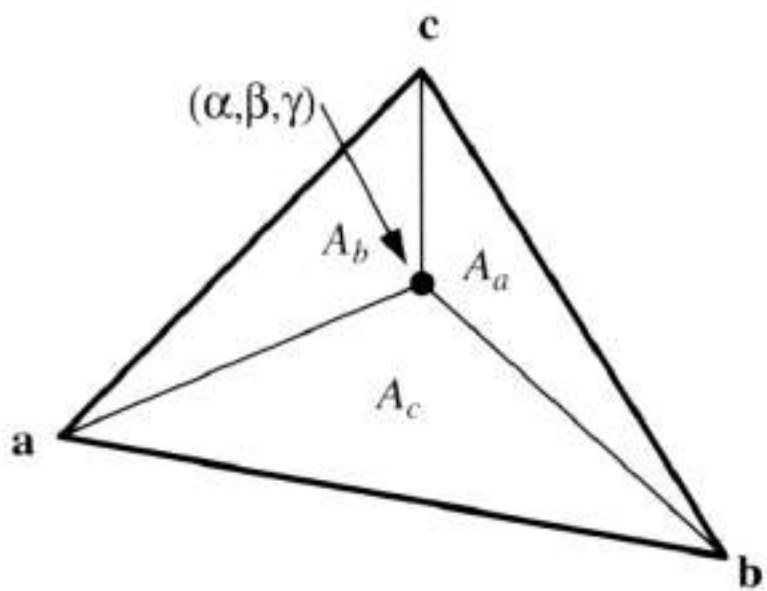
Barycentric coordinates

- Algebraic view: linear combination

$$w_0 = 1 - w_1 - w_2 \Rightarrow$$
$$\mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

- Geometric view: relative areas

$$w_0 = A_0/A \quad w_1 = A_1/A \quad w_2 = A_2/A$$



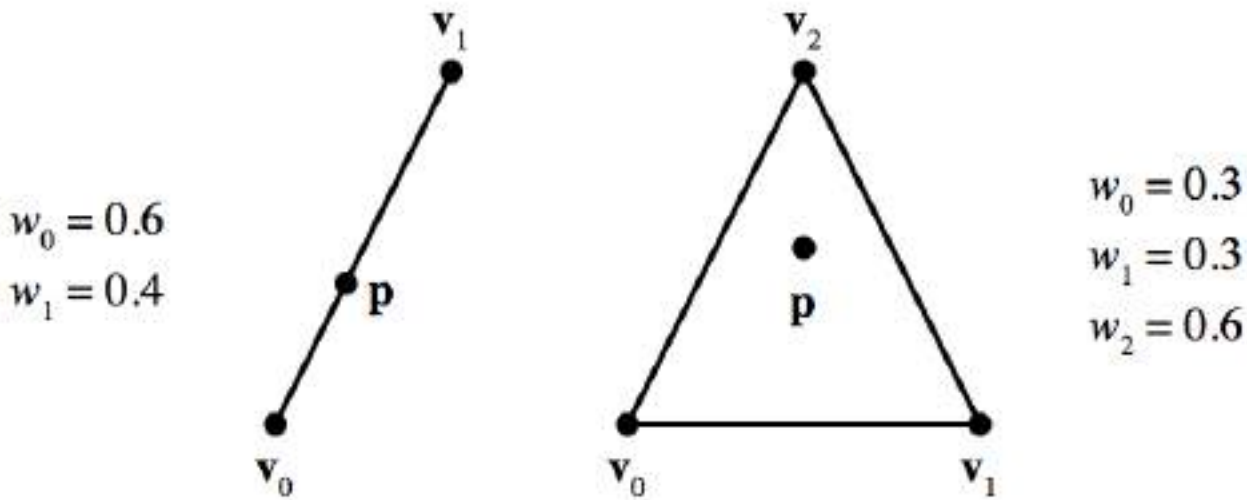
[Shirley 2000]

© 2020 Fabio Pellacini

Barycentric coordinates

- General concept for points, lines, triangles, tetrahedra
- Can write points as linear combination of vertices

$$\mathbf{p} = \sum_{i=0}^{n-1} w_i \mathbf{v}_i \qquad \sum_{I=0}^{n-1} w_i = 1 \qquad w_i \geq 0$$



© 2020 Fabio Pellacini

Triangle Meshes

© 2017 Fabio Pellacini and Steve Marschner

Triangle meshes

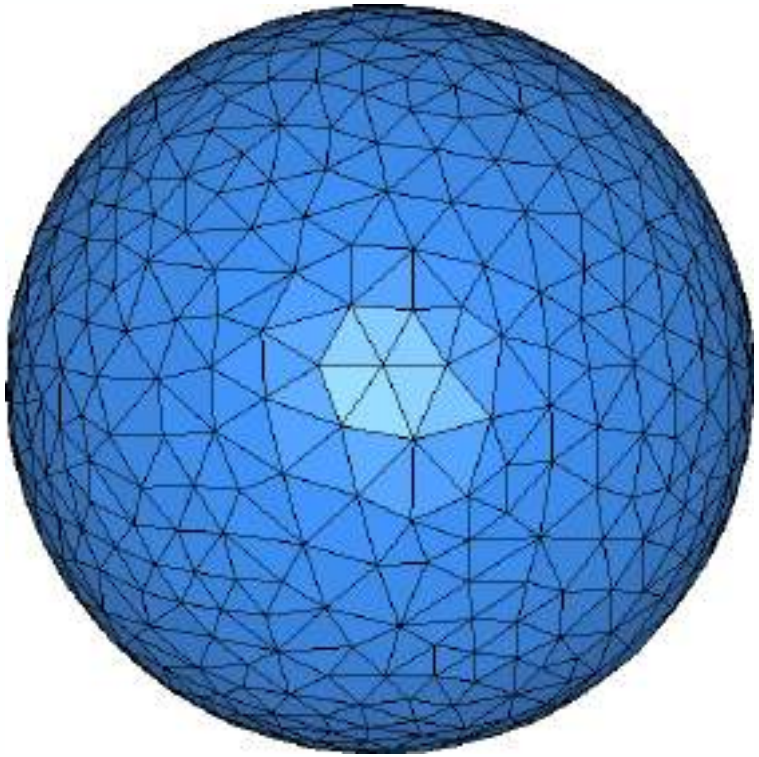
- A set triangles in 3D space connected together to form a surface
- Most common shape used in 3D graphics applications
- Geometrically, a mesh is a piecewise planar surface: it is planar except at the edges where triangles join
- Often, it's a piecewise planar approximation of a smooth surface
  - in this case the creases between triangles are artifacts—we don't want to see them

© 2020 Fabio Pellacini

Approximate geometry

[Andrzej Barabasz]

spheres



approximate sphere

[Rineau & Yvinec CGAL manual]

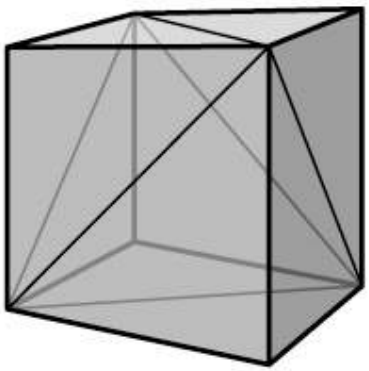
© 2020 Fabio Pellacini

Approximate geometry

- More triangles, better approximation

© 2020 Fabio Pellacini

A small triangle mesh



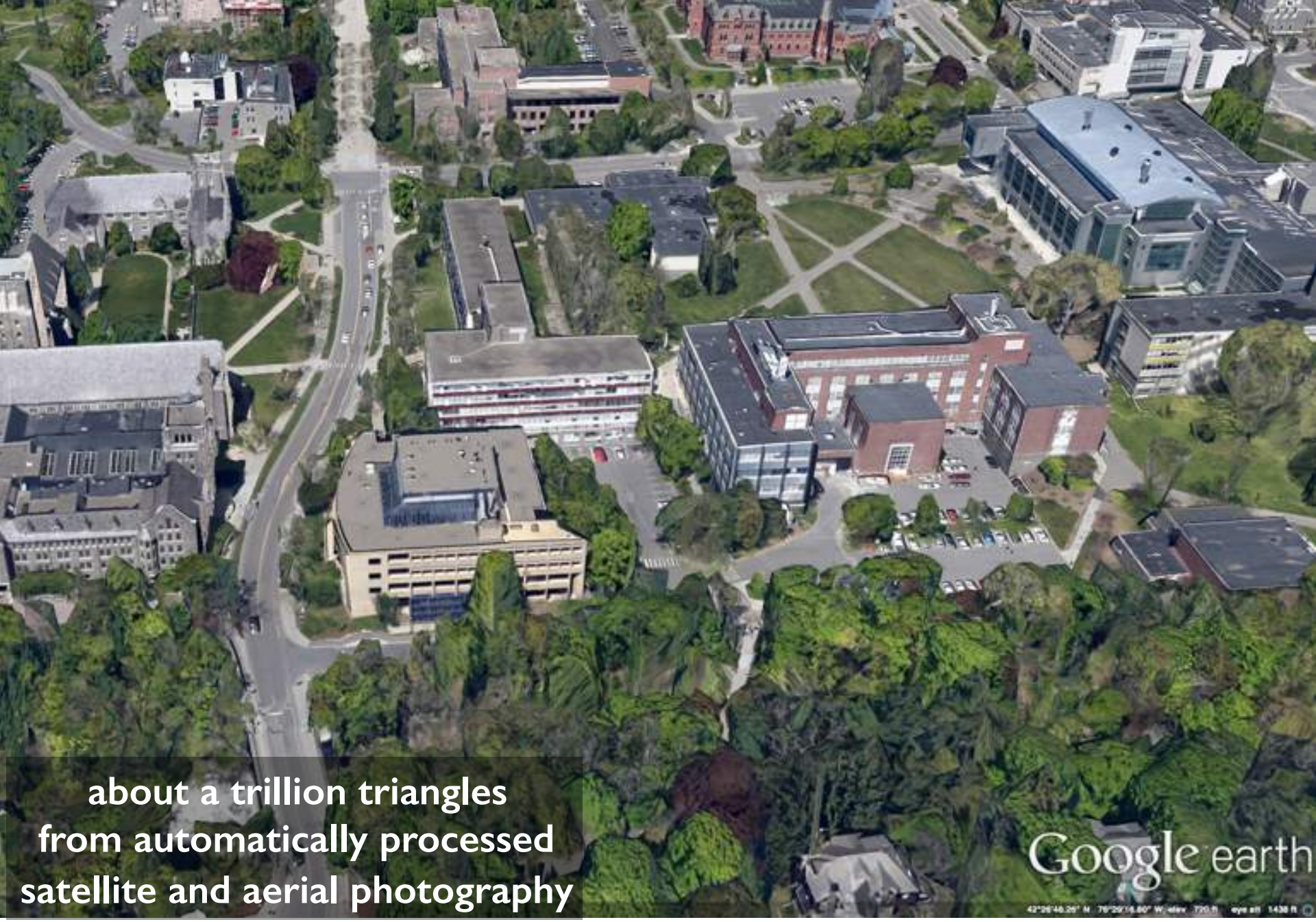
12 triangles, 8 vertices

© 2020 Fabio Pellacini

A large mesh

10 million triangles
from a high-resolution
3D scan

[Traditional Thai sculpture—scan by XYZRGB, inc., image by MeshLab]

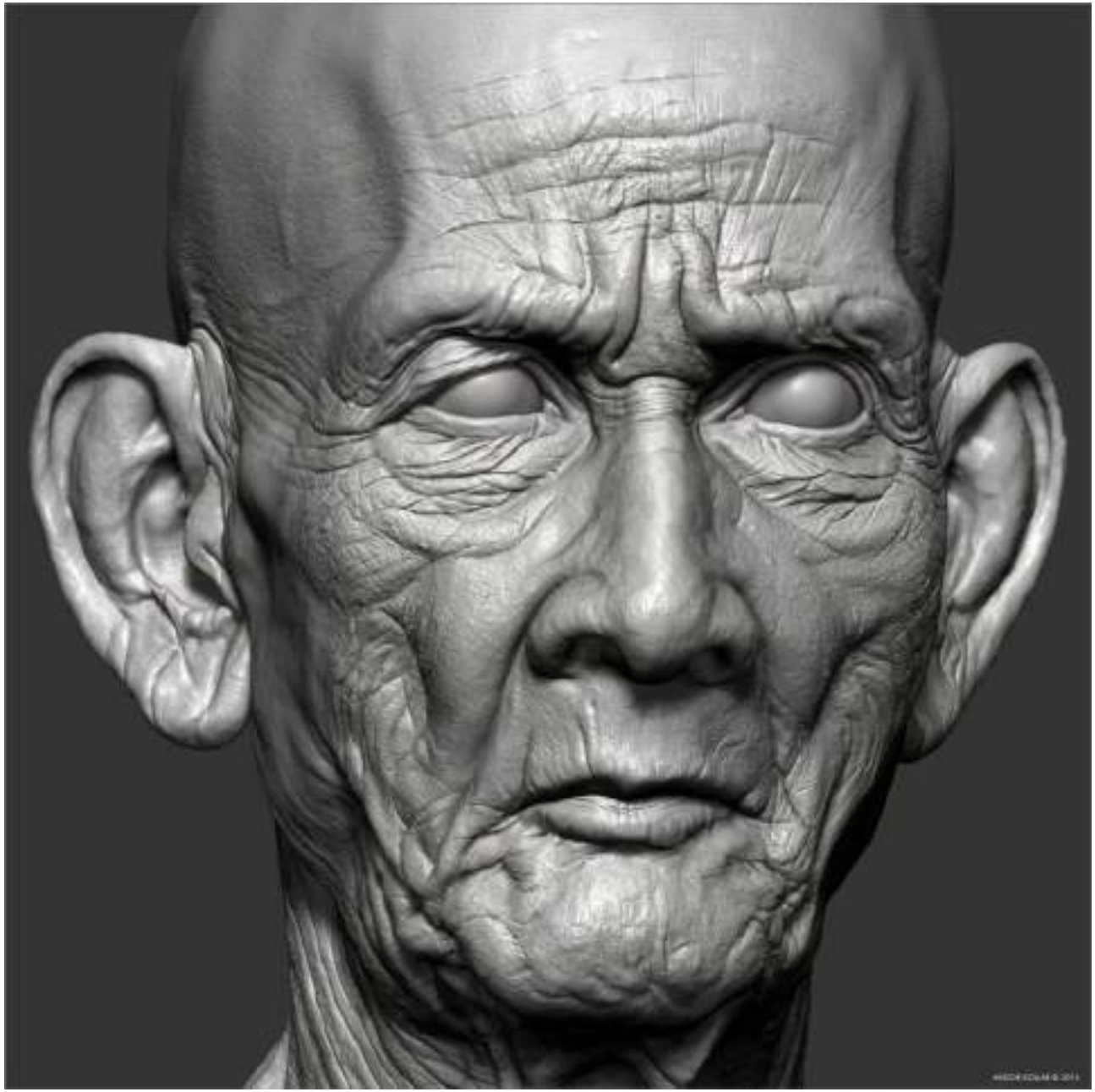


about a trillion triangles
from automatically processed
satellite and aerial photography

© 2020 Fabio Pellacini

[Andor Kollar]

© 2020 Fabio Pellacini

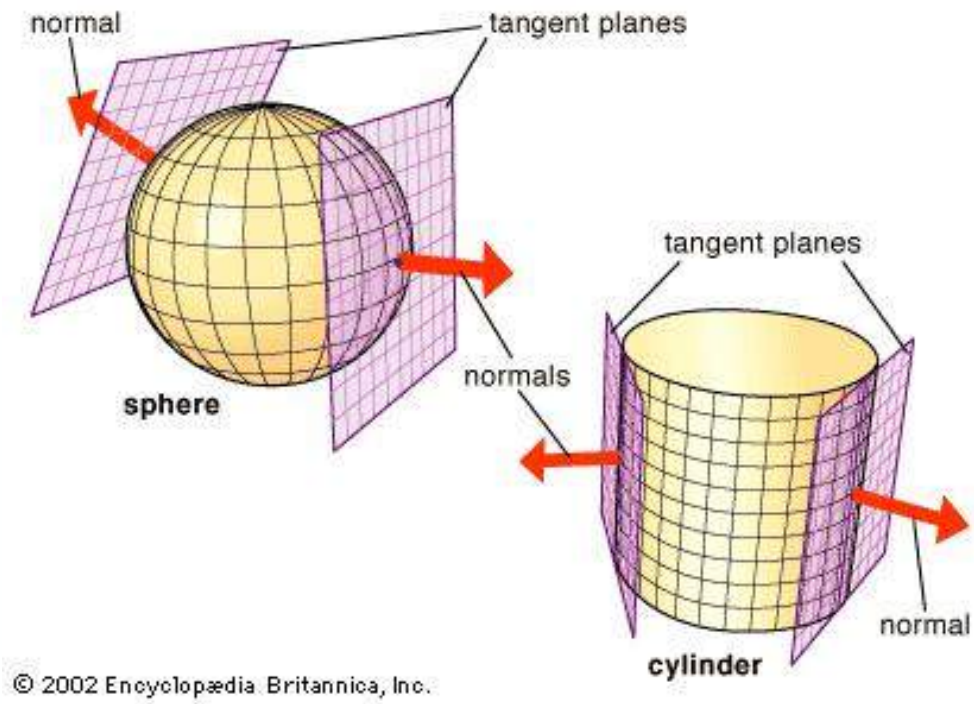


[Andor Kollar]

© 2020 Fabio Pellacini

Normals and tangents

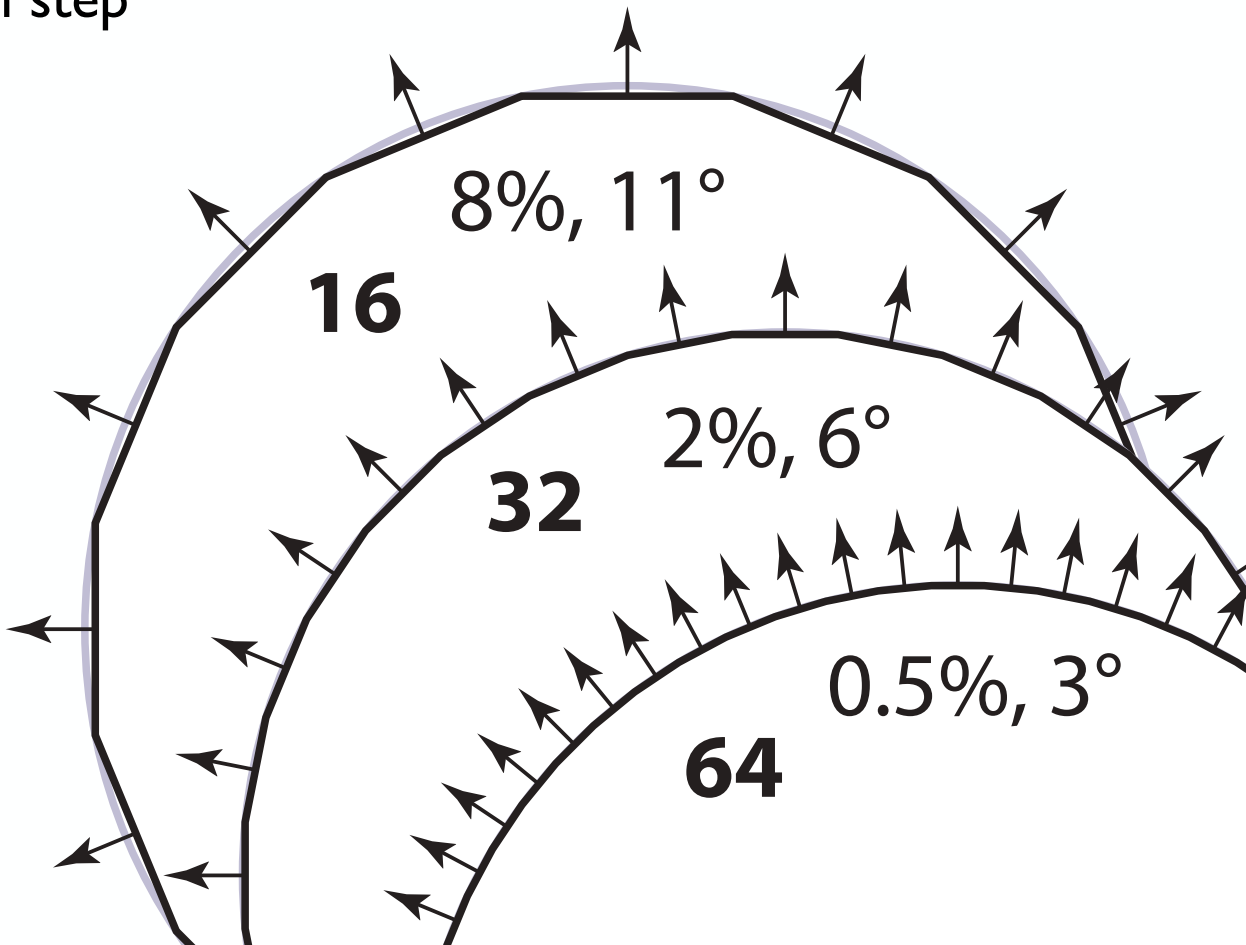
- Tangent plane: at a point on a smooth surface in 3D, there is a unique plane tangent to the surface, called the tangent plane
- Normal vector: vector perpendicular to the tangent plane
  - only unique for smooth surfaces (not at corners, edges)



© 2020 Fabio Pellacini

Approximation quality

- Approximating circle with increasingly many segments
- Max error in position error drops by factor of 4 at each step
- Max error in normal only drops by factor of 2

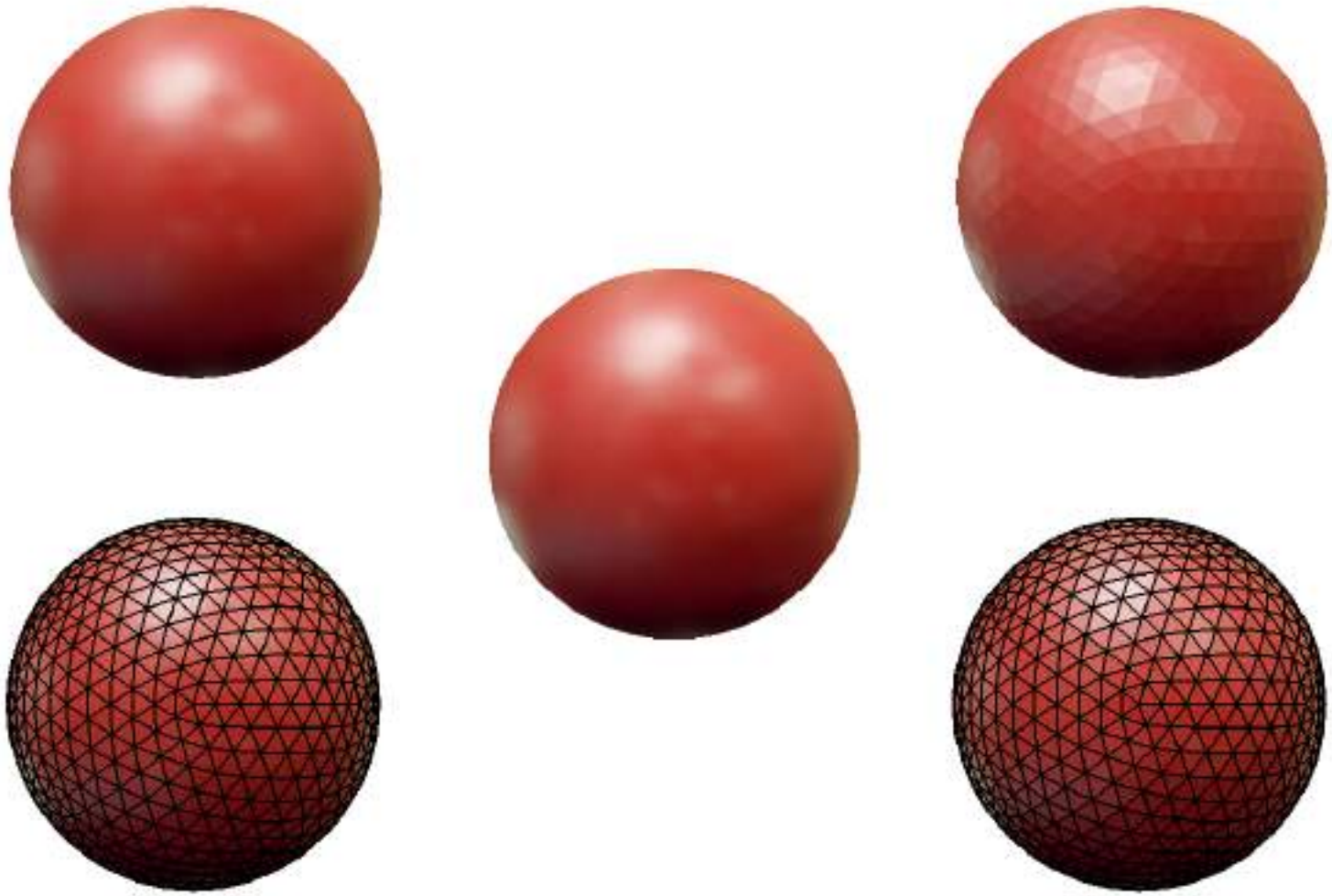


Approximation quality

- Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases
  - for mathematicians: error is $O(h^2)$
- But the surface normals don't converge so well
  - normal is constant over each triangle, with discontinuous jumps across edges
  - for mathematicians: error is only $O(h)$
- Better: store the "real" normal at each vertex, and interpolate to get normals that vary gradually across triangles

© 2020 Fabio Pellacini

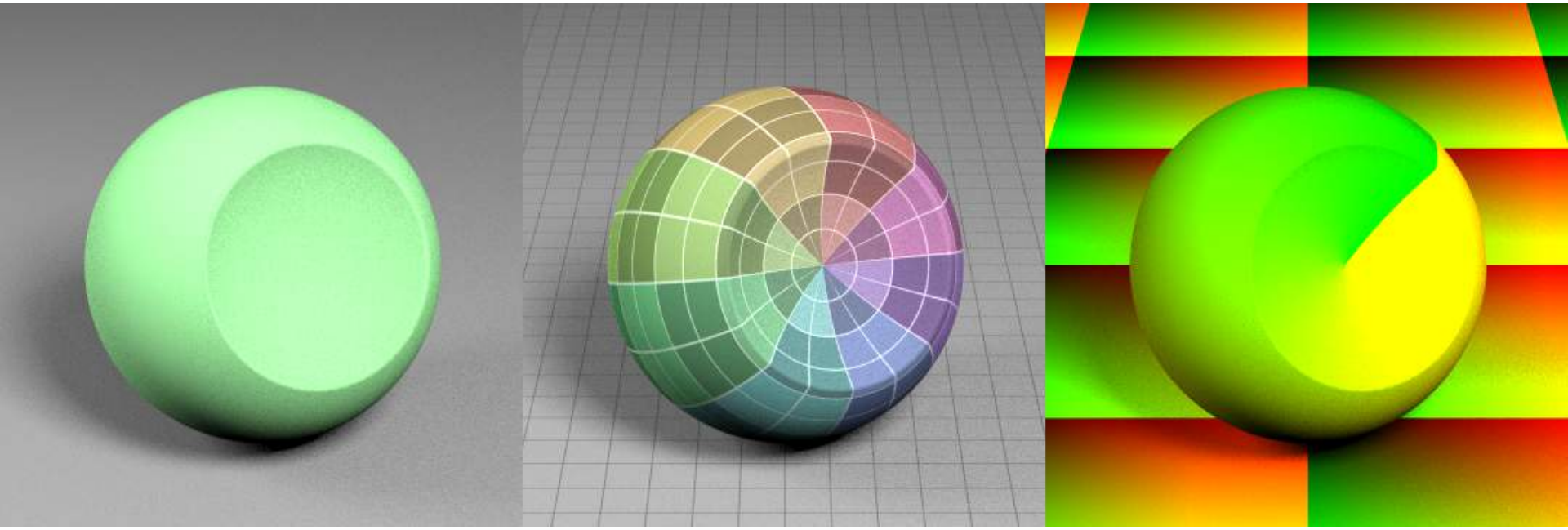
Facet vs. vertex normals



© 2020 Fabio Pellacini

Textures

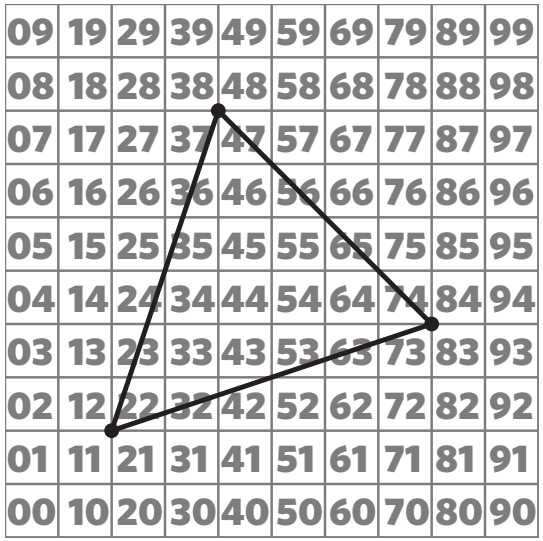
- Surface material parameters can be associated with each triangle
  - so we need ~ 1 triangle/pixel for full color resolution
- Idea: define materials by "pasting" images onto the surface
- We'll discuss in more details later

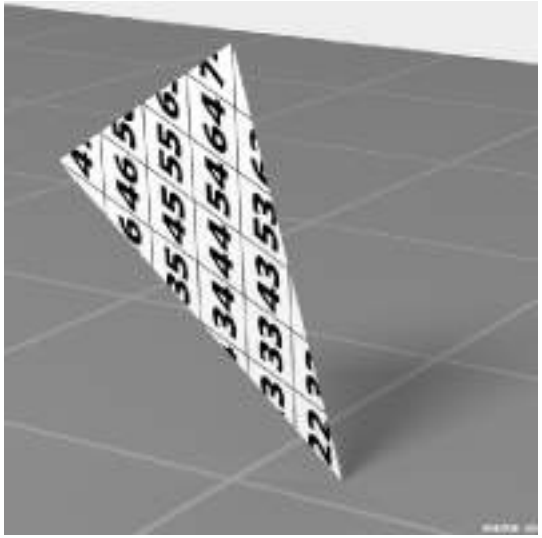


© 2020 Fabio Pellacini

Texture coordinates

- A surface in 3D is a two-dimensional entity
- Sometimes we need 2D coordinates for points on the surface
- Defining these coordinates is *parameterizing* the surface
- For triangle meshes, specify 2D coordinates (in $[0,1]^2$) at the vertices and barycentric interpolate in the triangle





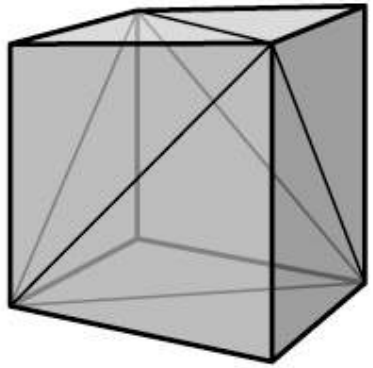
© 2020 Fabio Pellacini

Mesh Representations

© 2017 Fabio Pellacini and Steve Marschner

Topology vs. geometry

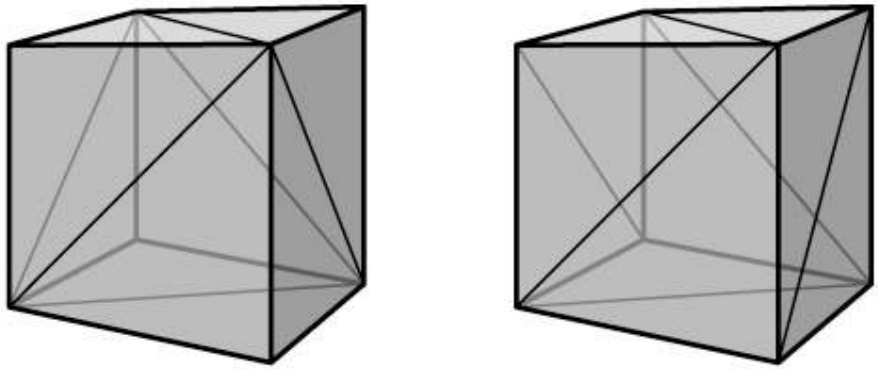
- Mesh geometry: where the triangles are in 3D space
- Mesh topology: how the triangles are connected
  - ignoring the positions entirely



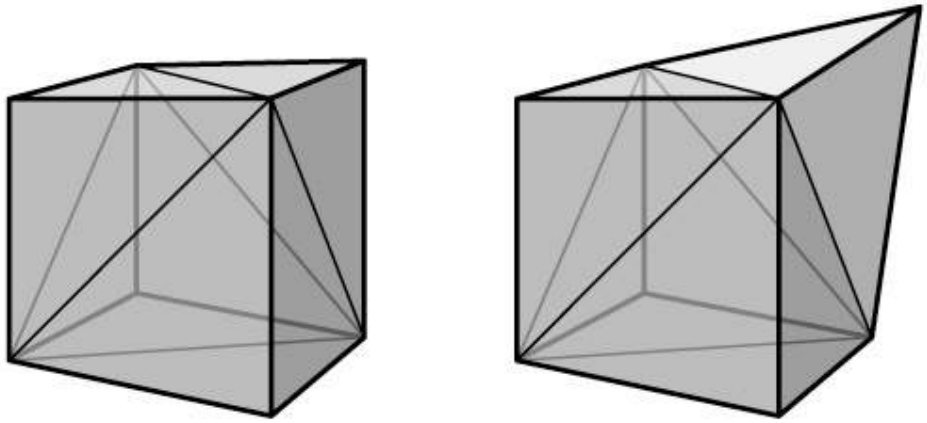
© 2020 Fabio Pellacini

Topology vs. geometry

- same geometry, different mesh topology:



- same mesh topology, different geometry:



© 2020 Fabio Pellacini

Mesh representations

- Compactness
- Efficiency for rendering
  - enumerate all triangles as triples of 3D points
- Efficiency of adjacency queries
  - all vertices of a triangle
  - all triangles around a vertex
  - neighboring triangles of a triangle
  - need depends on application (rendering, subdivs, modeling)

© 2020 Fabio Pellacini

Mesh representations

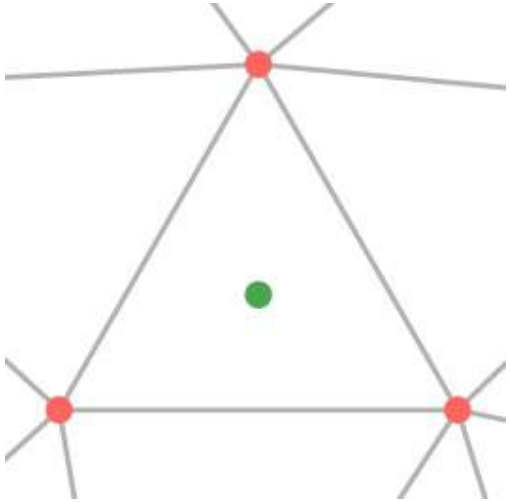
- Separate triangles
- Indexed triangle set
  - shared vertices
- Triangle strips and triangle fans
  - compression schemes for fast transmission
- Triangle-neighbor data structure
  - supports adjacency queries
- Half-edge data structure
  - supports general polygon meshes

© 2020 Fabio Pellacini

Separate triangles

- Store each triangle independently

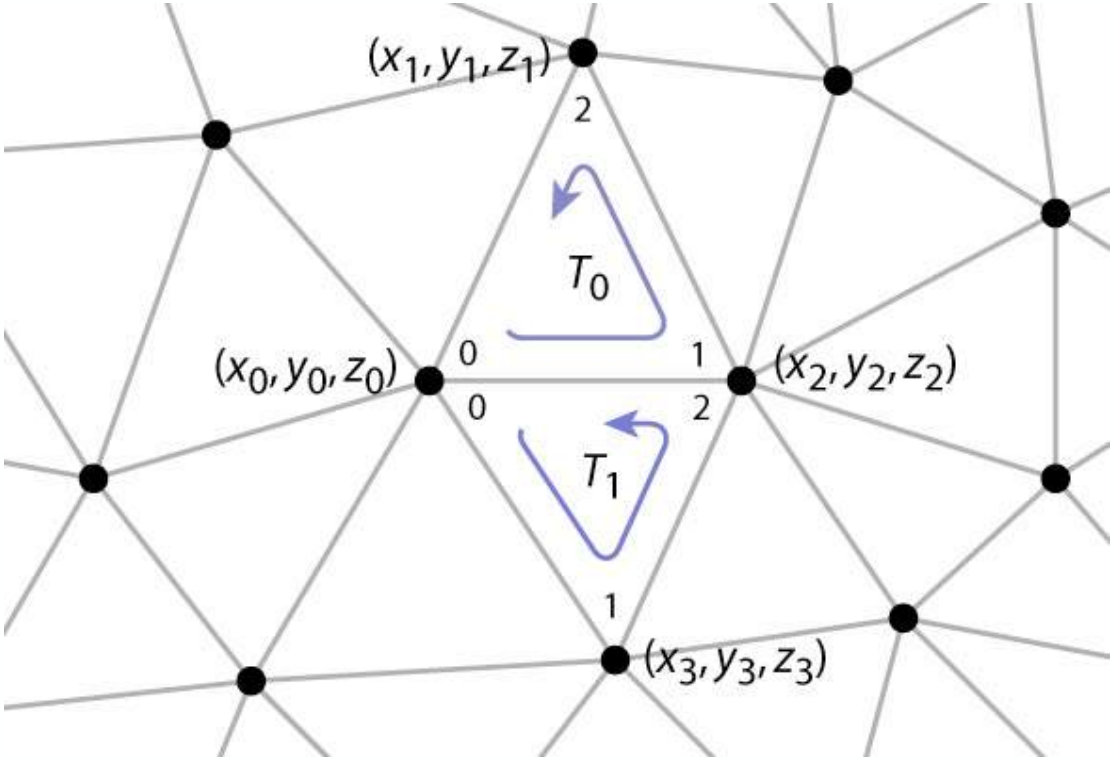
```
struct mesh {
  vec3f pos[nT][3];
};
```



© 2020 Fabio Pellacini

Separate triangles

	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮



© 2020 Fabio Pellacini

Separate triangles

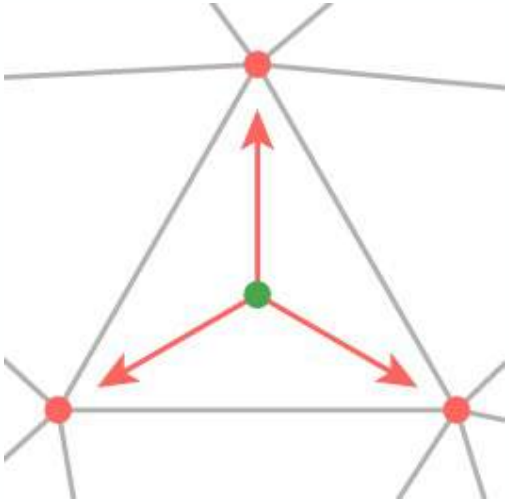
- Array of triples of points
  - float[n_T][3][3]: about 72 bytes per vertex
  - 2 triangles per vertex (on average)
- Various problems
  - wastes space (each vertex stored 6 times)
  - cracks due to roundoff
  - difficulty of finding neighbors at all

© 2020 Fabio Pellacini

Indexed triangles

- Store each vertex once
- Each triangle points to its three vertices

```
struct mesh {
  vec3i tris[nT];
  vec3f pos[nV];
};
```

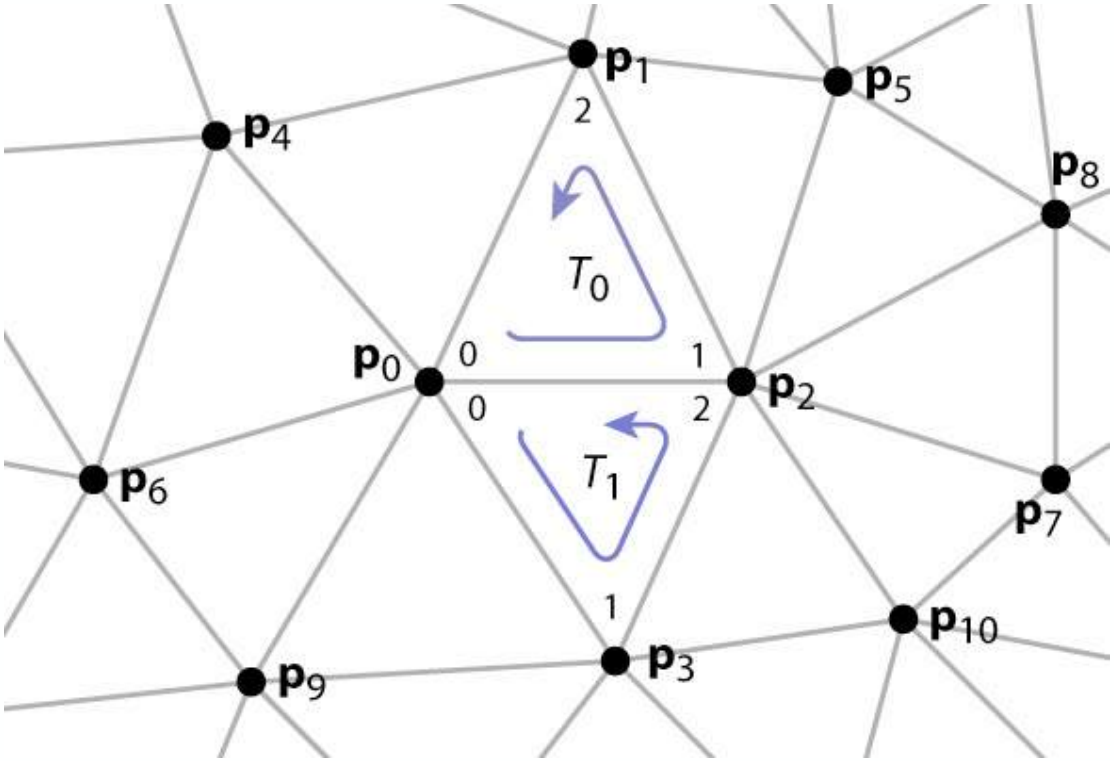


© 2020 Fabio Pellacini

Indexed triangles

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
	⋮



© 2020 Fabio Pellacini

Indexed triangles

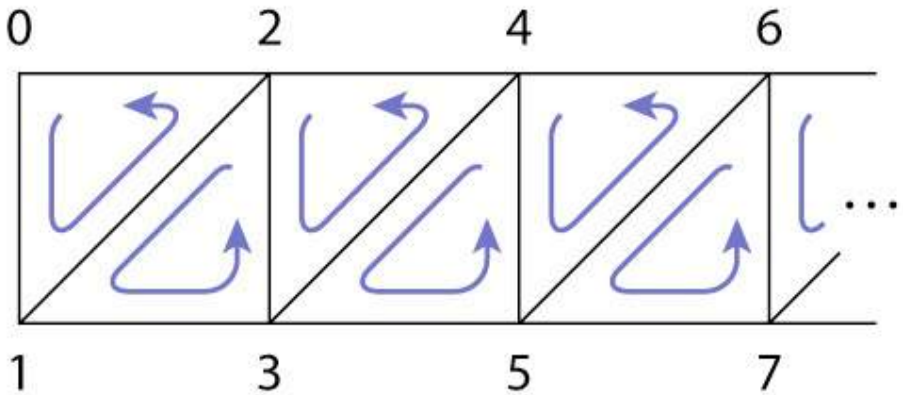
- Array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
- Array of triangle indices
  - int[n_T][3]: 12 bytes per triangle
  - 2 triangles per vertex on average: 24 bytes per vertex
- Total storage: 36 bytes per vertex (factor of 2 savings)
- Represents topology and geometry separately
- Finding neighbors is well defined

© 2020 Fabio Pellacini

Triangle strips

- Compress triangle lists by skipping repeated indices
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing the second and third vertex of the previous triangle
  - for long strips, this requires about one index per triangle

```
struct mesh {
  int strips[nT][];
  float pos[nV][3];
};
```

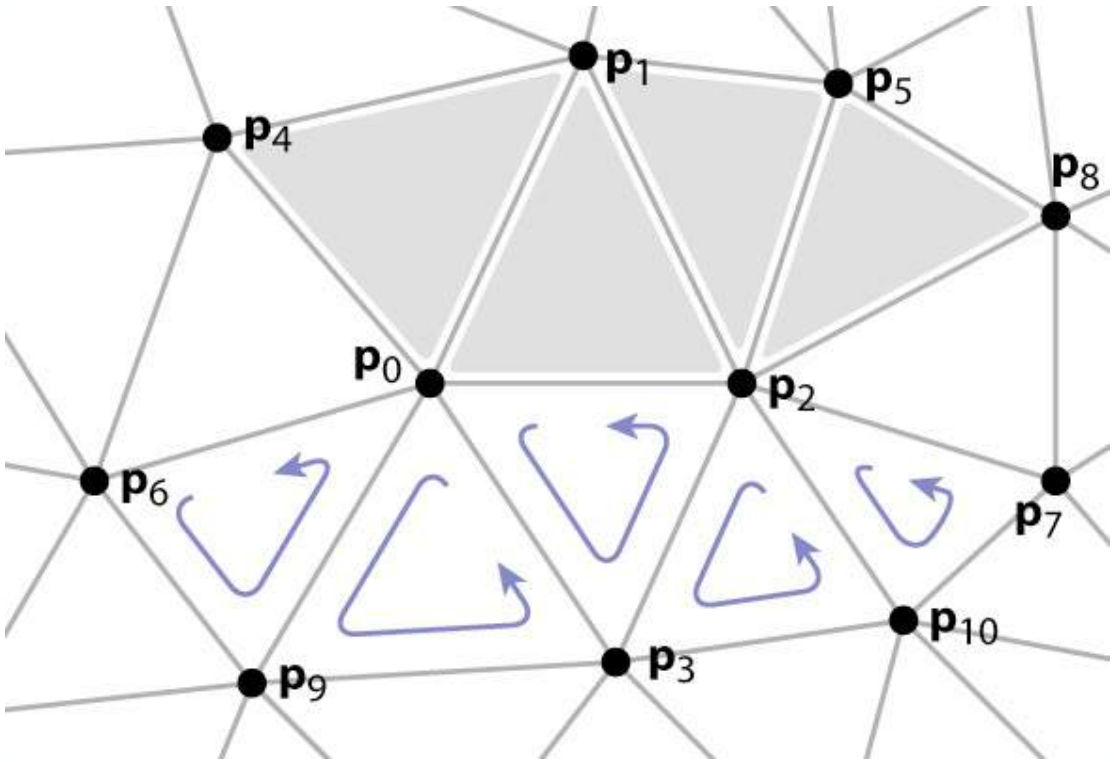


© 2020 Fabio Pellacini

Triangle strips

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tStrip[0]	6, 0, 4, 1, 2, 5, 8
tStrip[1]	6, 9, 0, 3, 2, 10, 7
	⋮



© 2020 Fabio Pellacini

Triangle strips

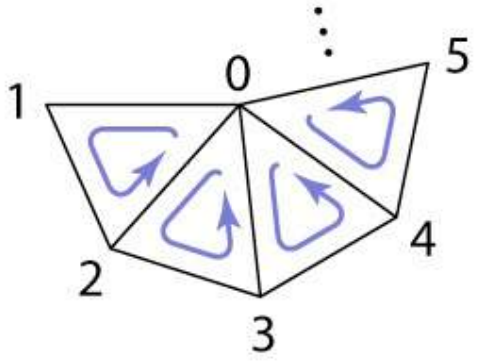
- Array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
- Array of index lists
  - int[n_S][variable]: (2 + n)x4 bytes per strip
  - on average, (1 + ε) indices per triangle (assuming long strips)
- Total is 20 bytes per vertex (limiting best case)
  - factor of 3.6 over separate triangles; 1.8 over indexed mesh

© 2020 Fabio Pellacini

Triangle fans

- Same idea as triangle strips, but keep oldest rather than newest
  - every sequence of three vertices produces a triangle
  - for long fans, this requires about one index per triangle
- Memory considerations exactly the same as triangle strip

```
struct mesh {
  int fans[nT][];
  float pos[nV][3];
};
```



© 2020 Fabio Pellacini

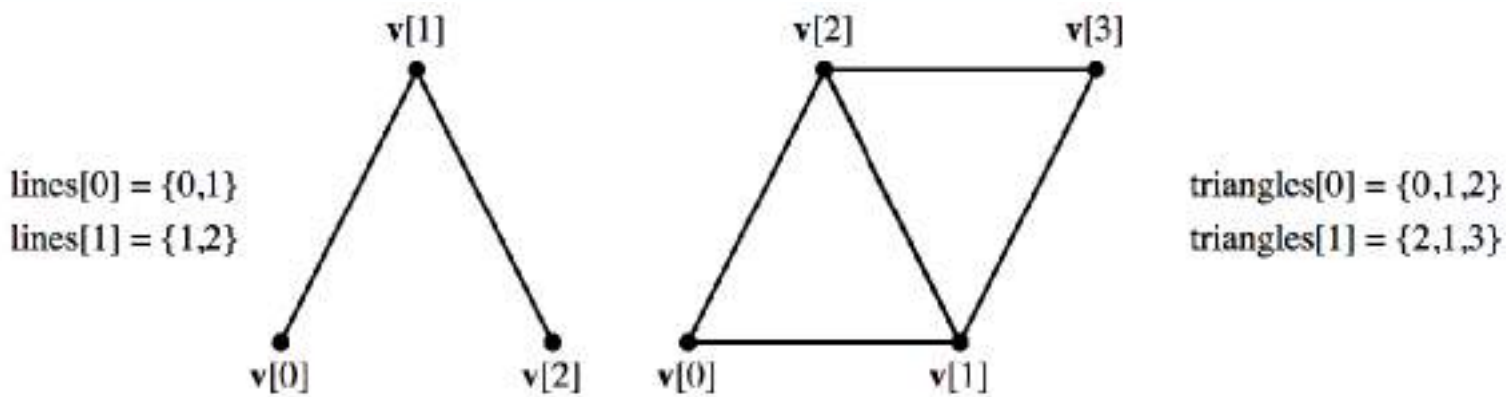
Data on meshes

- Often need to store additional information besides just the geometry
- Can store additional data at faces, vertices, or edges
- Store on faces for discontinuous data, e.g. facet colors
- Information about creases stored at edges
- Quantities that vary continuously (without sudden changes, or discontinuities) gets stored at vertices

© 2020 Fabio Pellacini

Indexed Mesh Data Structure

- Indexed triangles is the most used representation
  - efficient for rendering and many geometry operations
  - can easily add/remove triangles and vertices
- For GPU execution: store data on vertices
- In modeling applications: store data on vertices and faces
- Indexed representation used also for segments and tetrahedra



© 2020 Fabio Pellacini

Key types of vertex data

- Positions
  - at some level this is just another piece of data
  - position varies continuously between vertices
- Surface normals
  - when a mesh is approximating a curved surface, store normals at vertices
- Texture coordinates
  - 2D coordinates that tell you how to paste images on the surface

© 2020 Fabio Pellacini

Indexed Mesh Data Structure

- A typical implementation uses separate arrays for each vertex data
  - this allows to only store data if present
- Element indices are held in either an integer array or a typed array (as here) for easier accessing

```
struct shape {
  vector<vec3f> pos;       // vertex positions
  vector<vec3f> norm;      // vertex normals
  vector<vec2f> texcoord;  // vertex texture coordinates

  vector<int>   points;    // point indices
  vector<vec2i> lines;     // line indices
  vector<vec3i> triangles; // triangle indices
}
```


© 2020 Fabio Pellacini

Computing vertex normals

- Ideally, store normals from the "true" geometry
- Practically, "true" geometry is not available most of the time, e.g. in hand modeling or 3D scanning
- Compute the normal $\mathbf{n}_i$ of vertex i as the average the normals $\mathbf{n}_f$ of the triangles f that are adjacent to i, weighted by the triangles areas A_f

$$\mathbf{n}_i = \frac{\sum_{f \in N_i} A_f \mathbf{n}_f}{\sum_{f \in N_i} A_f}$$

- Implement this by looping over faces and updating vertex normals, which is $O(n)$, instead of a trivial implementation of the equation above which is $O(n^2)$

© 2020 Fabio Pellacini

Computing vertex normals

```
vector<vec3f> compute_normals(const vector<vec3f>& positions,
                              const vector<vec3i>& triangles) {

  // initialize normals to 0
  auto normals = vector<vec3f>(pos.size(), {0,0,0});
  // accumulate triangle normals
  for(auto t : triangles) {
    auto n = triangle_normal(positions[t.x],
      positions[t.y], positions[t.z]);
    auto a = triangle_area(positions[t.x],
      positions[t.y], positions[t.z]);
    normals[t.x] += a * n; normals[t.y] += a * n;
    normals[t.z] += a * n;
  }
  // normalize result
  for(auto& n : norm) n = normalize(n);
  return norm;
}
```


© 2020 Fabio Pellacini

Interpolating vertex data

- Interpolation of vertex data follows barycentric coordinates
  - for normals we normalize at the end

$$\mathbf{p} = w_0\mathbf{p}_0 + w_1\mathbf{p}_1 + w_2\mathbf{p}_2 \quad \mathbf{n} = \frac{w_0\mathbf{n}_0 + w_1\mathbf{n}_1 + w_2\mathbf{n}_2}{|w_0\mathbf{n}_0 + w_1\mathbf{n}_1 + w_2\mathbf{n}_2|}$$

```
vec3f normal(shape* shape, int element, vec2f uv) {
  auto t = shape->triangles[element];
  return normalize((1 - uv.x - uv.y) * shape->normals[t.x] +
           uv.x * shape->normals[t.y] +
           uv.y * shape->normals[t.z]);
}
vec3f position(shape* shape, int element, vec3f uv) {
  auto t = shape->triangles[element];
  return (1 - uv.x - uv.y) * shape->positions[t.x] +
         uv.x * shape->positions[t.y] +
         uv.y * shape->positions[t.z]);
}
```

Mesh Validity

© 2017 Fabio Pellacini and Steve Marschner

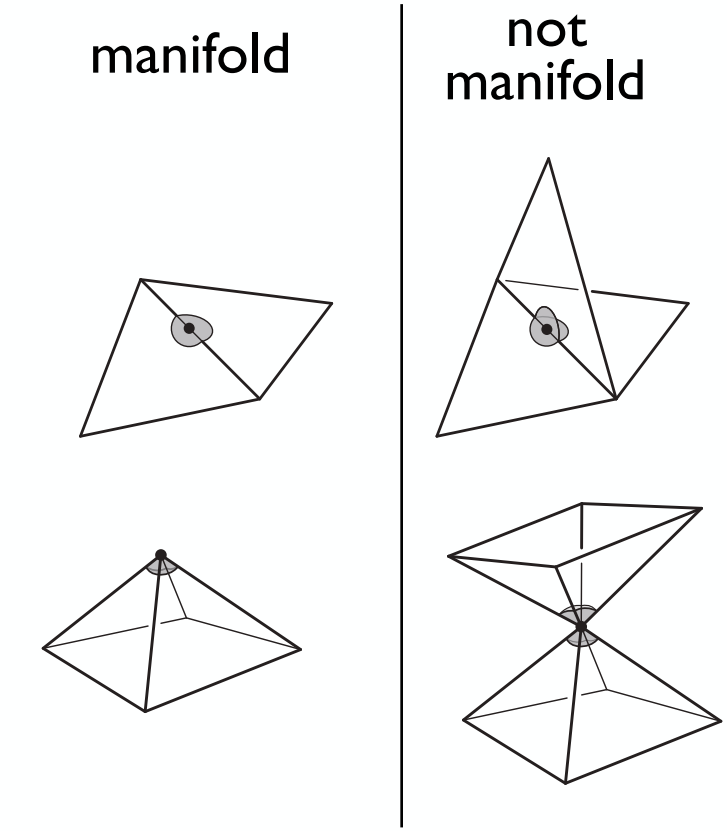
Validity of triangle meshes

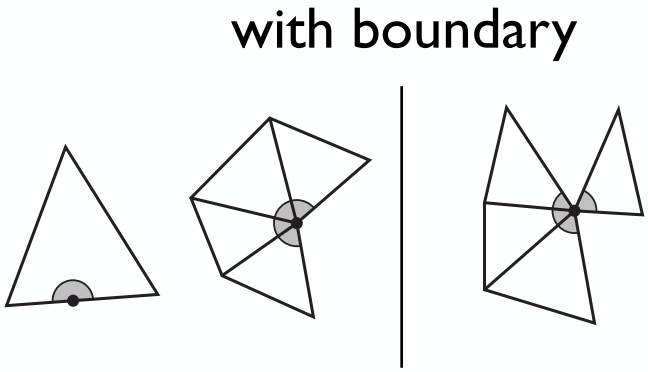
- In many cases we care about the mesh being able to bound a region of space nicely
- In other cases we want triangle meshes to fulfill assumptions of algorithms that will operate on them (and may fail on malformed input)
- Two completely separate issues:
  - topology: how the triangles are connected
  - geometry: where the triangles are in 3D space

© 2020 Fabio Pellacini

Topological validity

- Strongest property: be a manifold
  - this means that no points should be "special"
  - interior points are fine
  - edge points: each edge must have exactly 2 triangles
  - vertex points: each vertex must have one loop of triangles
- Slightly looser: manifold with boundary
  - weaken rules to allow boundaries

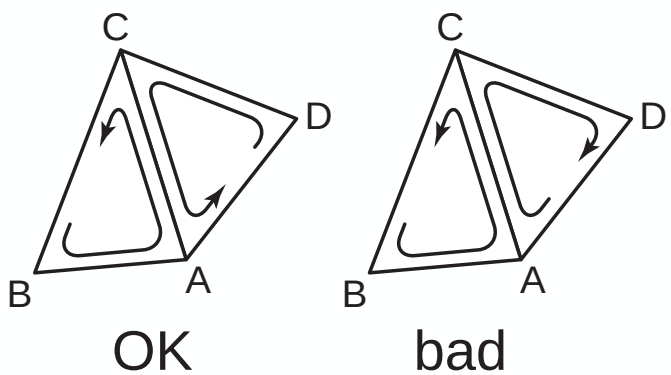


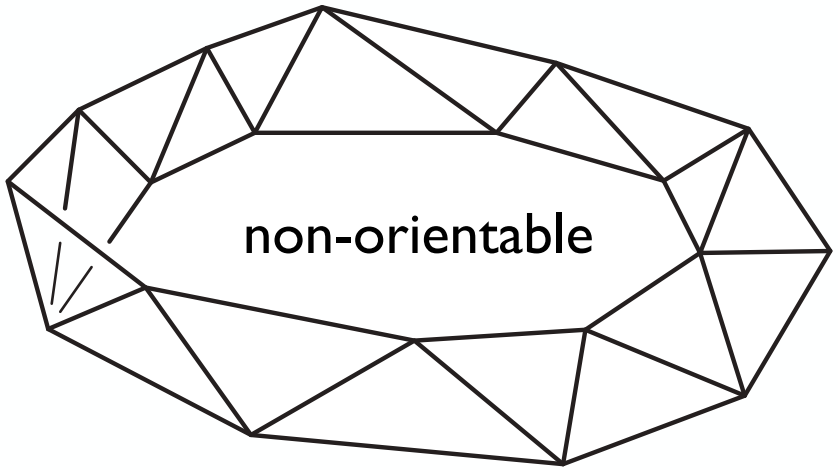


© 2020 Fabio Pellacini

Topological validity

- Consistent orientation
  - which side is the "front" or "outside" of the surface and which is the "back" or "inside?"
  - rule: you are on the outside when you see the vertices in counter-clockwise order
  - in mesh, neighboring triangles should agree about which side is the front!
  - caution: not always possible

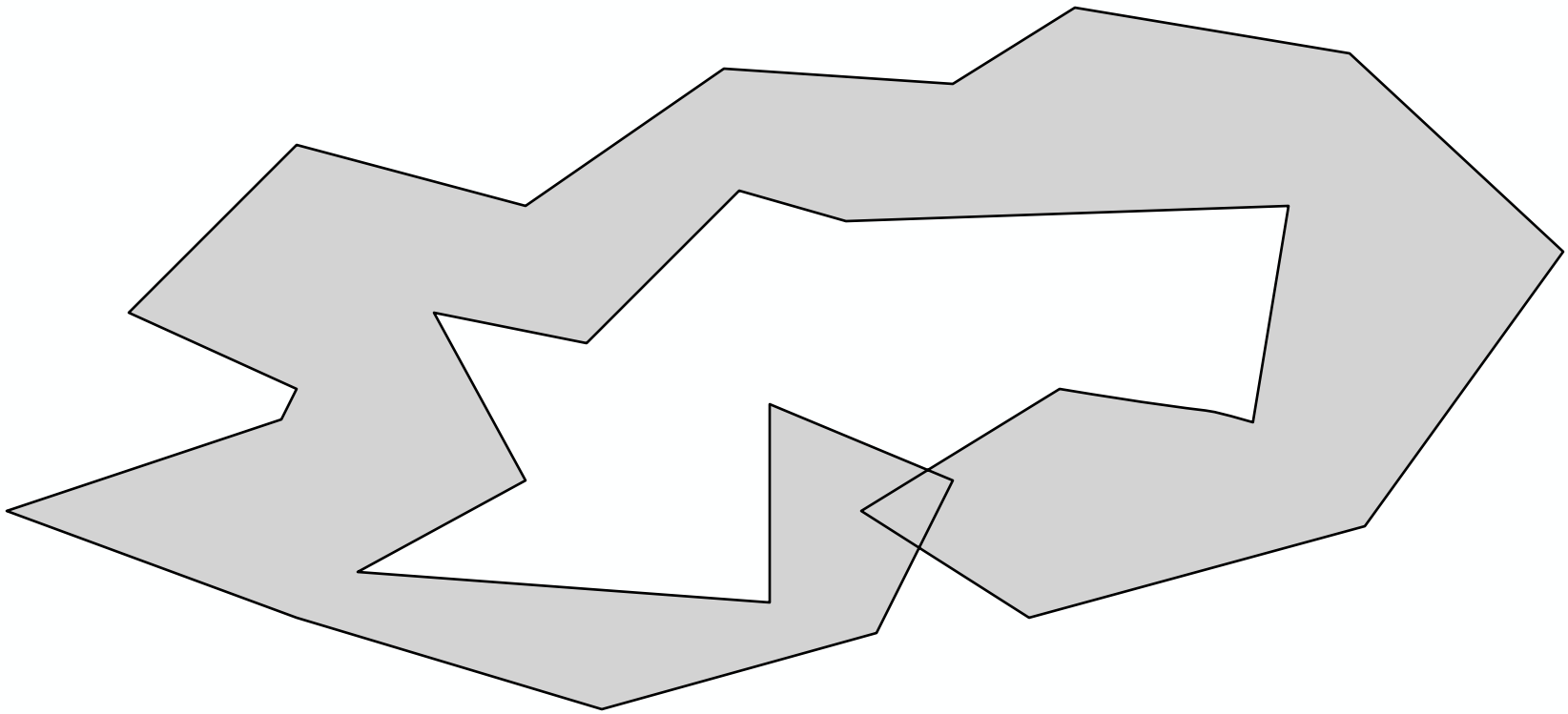




© 2020 Fabio Pellacini

Geometric validity

- Generally want non-self-intersecting surface
- Hard to guarantee in general
  - because far-apart parts of mesh might intersect

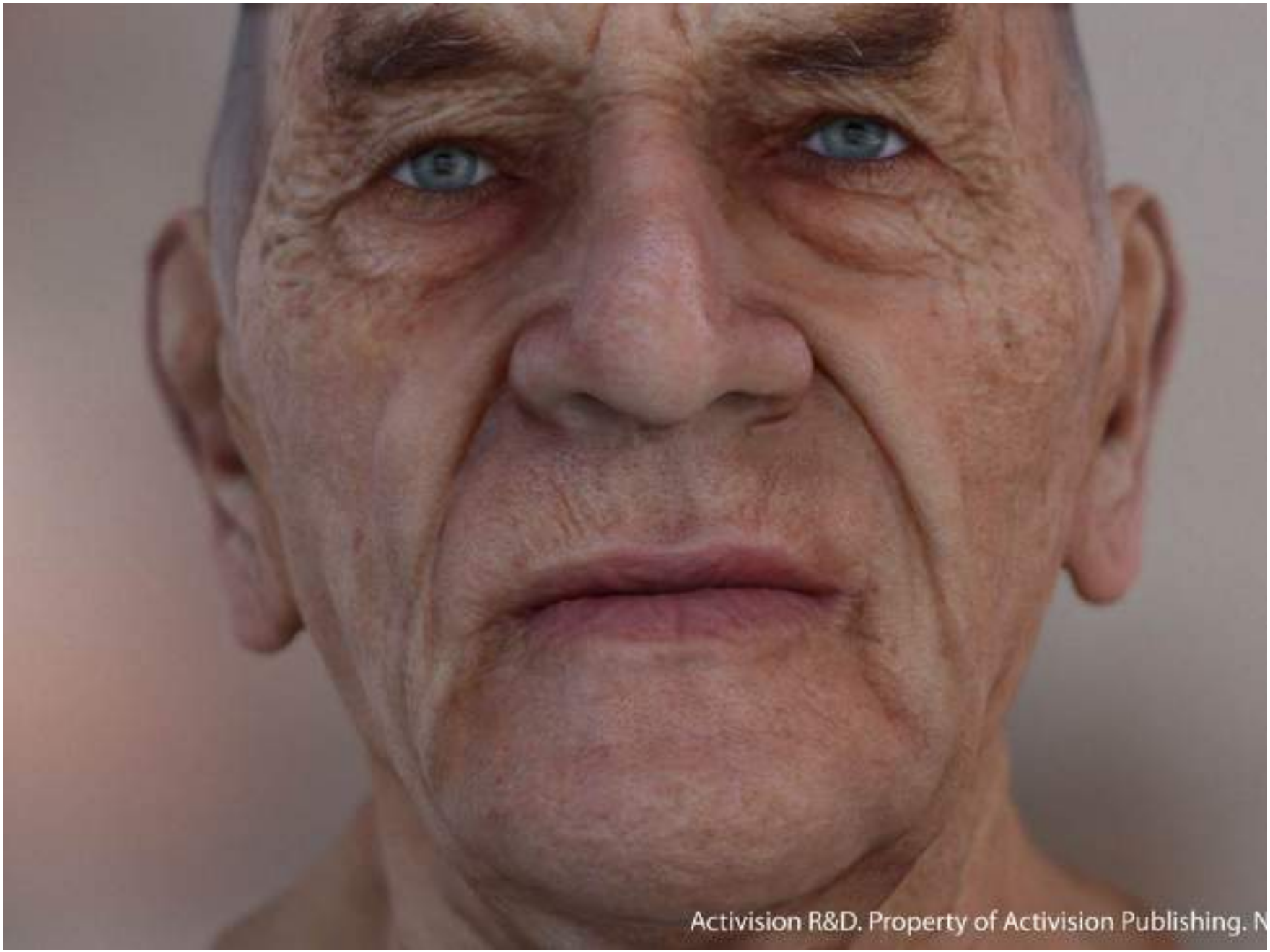


© 2020 Fabio Pellacini

Representing Materials

© 2017 Fabio Pellacini and Steve Marschner

© 2020 Fabio Pellacini

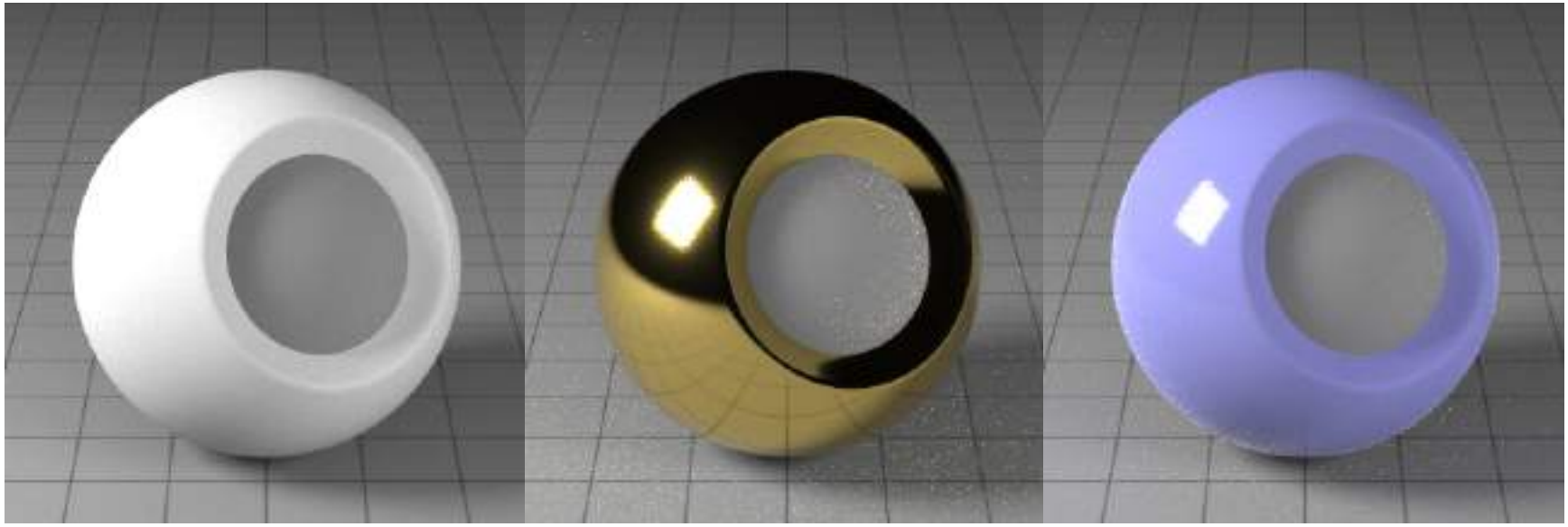


[Activision]

© 2020 Fabio Pellacini

Representing Materials

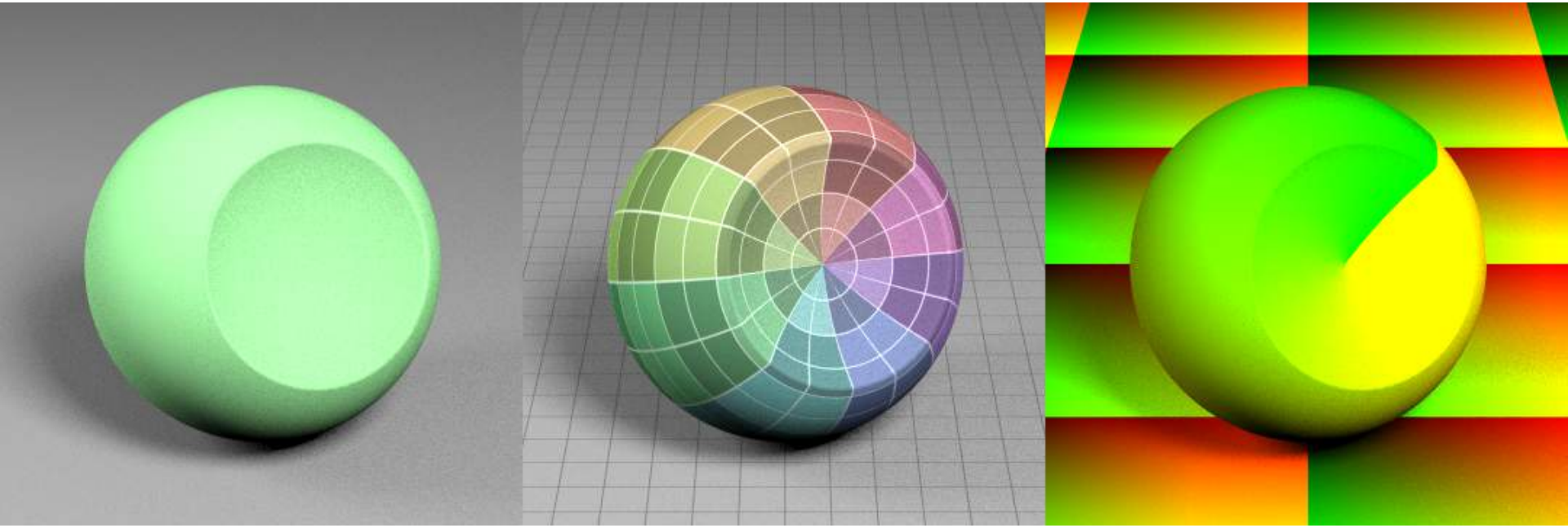
- Start with a simple model that covers plastics and metals
  - augment the model later for other material types
- Diffuse color: controls color for plastics
- Specular color : controls color for metals
- Specular roughness: controls highlight/reflection size and blur



© 2020 Fabio Pellacini

Representing Materials

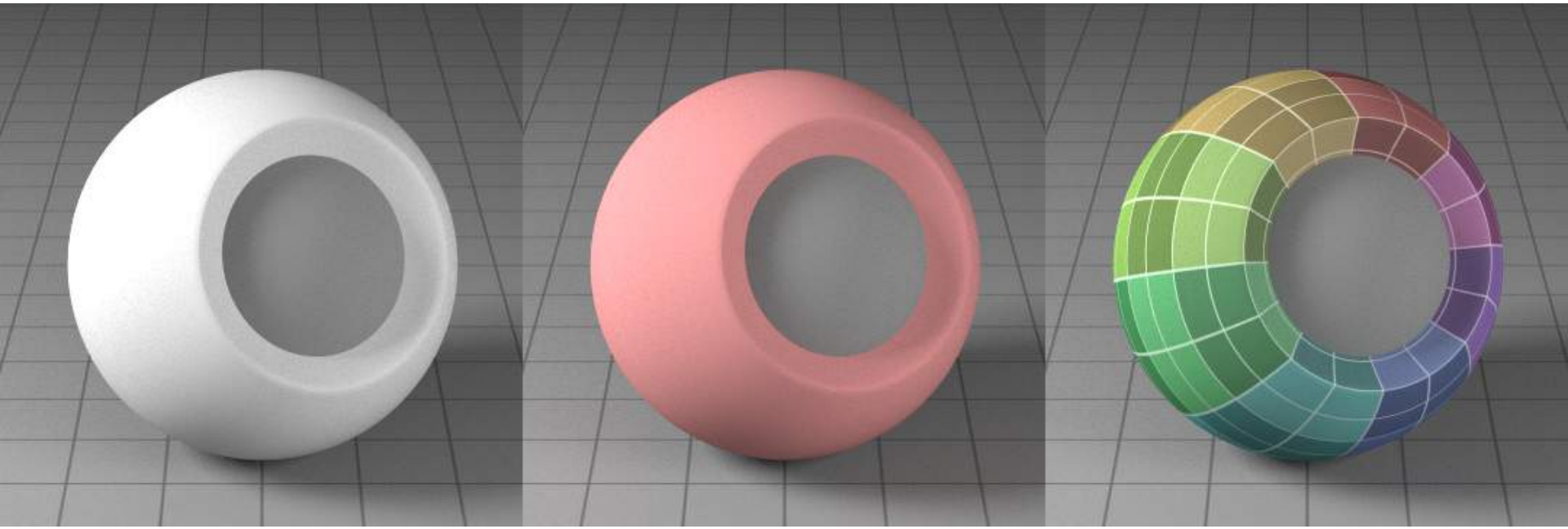
- Common to specify parameters both as constant and textures
  - take the product of the two



© 2020 Fabio Pellacini

Matte surfaces

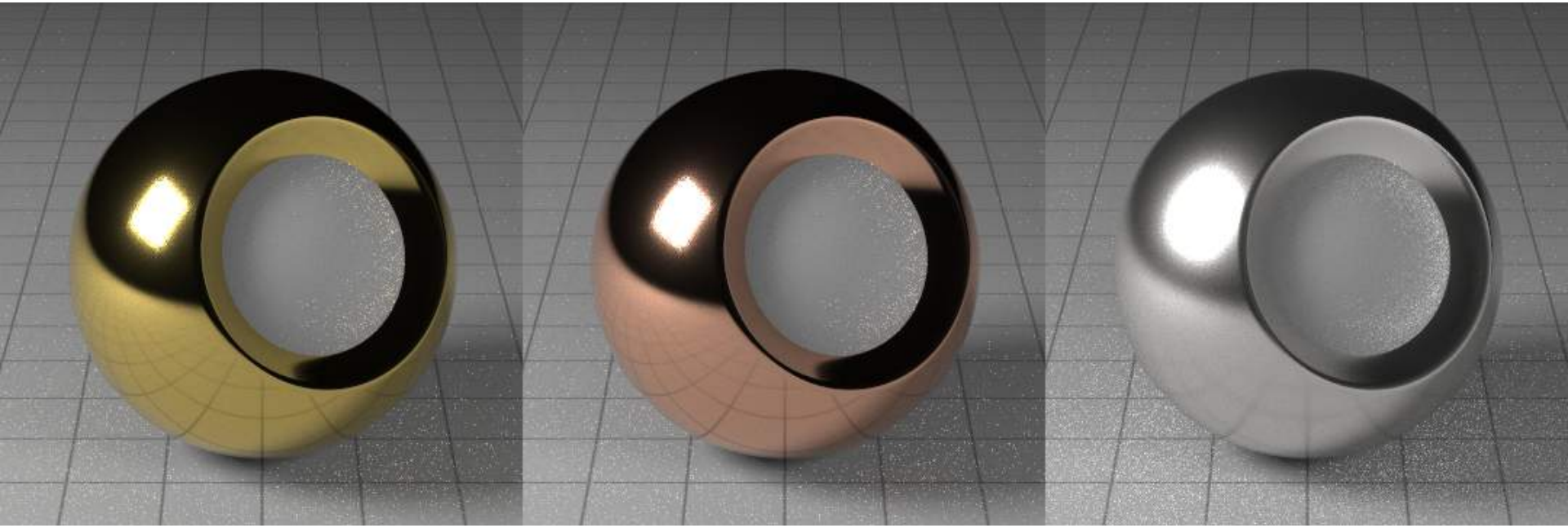
- Change color by changing the diffuse coefficient
- Apply textures to specify variations



© 2020 Fabio Pellacini

Metal surfaces

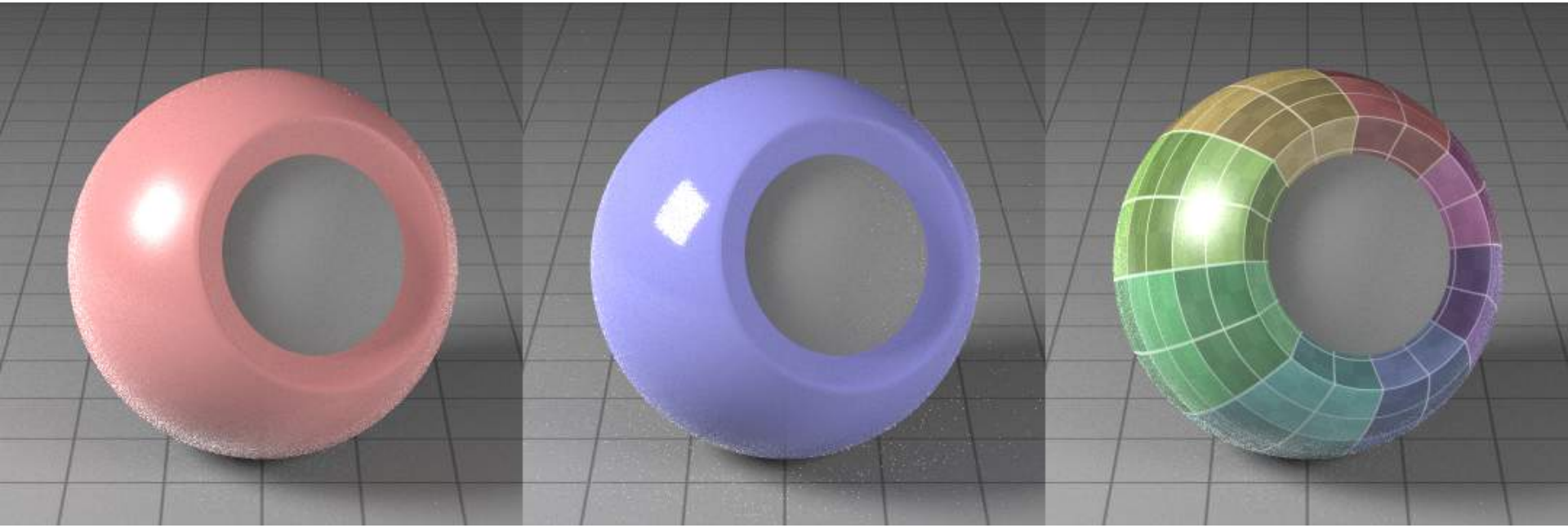
- Change color by changing the specular coefficient
- Change reflection blur by changing roughness



© 2020 Fabio Pellacini

Plastic surfaces

- Change color by changing diffuse coefficient
- Change highlights by changing roughness
  - set highlight color to a small value



© 2020 Fabio Pellacini

Representing Materials

- Legacy materials stored coefficients directly
- Store textures as pointers, so they can be shared between materials

```
struct texture {
  image<vec3f> hdr; // either hdr, e.g. for emission
  image<vec3b> ldr; // or ldr, e.g for colors
};

struct legacy_material {
  vec3f emission;         // emission
  vec3f diffuse;          // diffuse
  vec3f specular;         // specular
  float roughness;        // roughness
  texture* emission_tex;  // emission texture
  texture* diffuse_tex;   // diffuse texture
  texture* specular_tex;  // specular texture
  texture* roughness_tex; // roughness texture
};
```


© 2020 Fabio Pellacini

Representing Materials

- Current trends use one main color and weights to define behaviors

```
struct texture {
  image<vec3f> colorf;  // for color values
  image<vec3b> colorb;
  image<float> scalarf; // for scalar values
  image<byte>  scalarb;
};

struct material {
  vec3f emission;          // emission
  vec3f color;             // color
  float metallic;          // metallic
  float roughness;         // roughness
  texture* diffuse_tex;    // diffuse texture
  texture* metallic_tex;   // metallic texture
  texture* roughness_tex;  // roughness texture
};
```

Representing Lights

- No need for special objects: lights are just surfaces that emit light
- Stored as emission in the material

```
struct texture {
  image<vec3f> colorf;  // for color values
  image<vec3b> colorb;
  image<float> scalarf; // for scalar values
  image<vec3b> scalarb;
};
struct material {
  vec3f emission;          // emission
  vec3f color;             // color
  float metallic;          // metallic
  float roughness;         // roughness
  texture* emission_tex;   // emission texture
  texture* diffuse_tex;    // diffuse texture
  texture* metallic_tex;   // metallic texture
  texture* roughness_tex;  // roughness texture
};
```


© 2020 Fabio Pellacini

Textures

© 2017 Fabio Pellacini and Steve Marschner

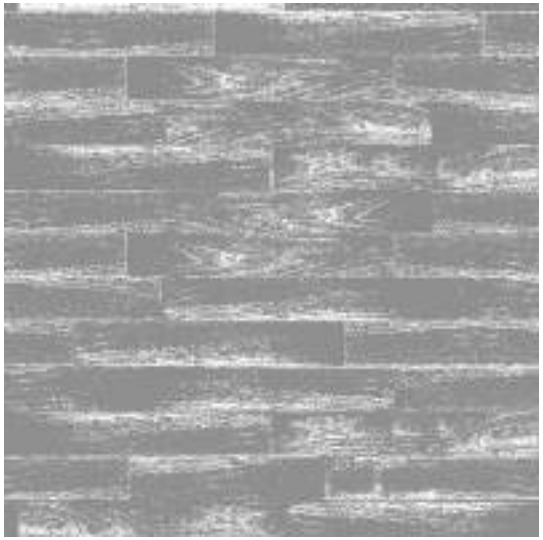
Texture mapping

- Objects have properties that vary across the surface
- Apply images to vary material parameters over the surface

© 2020 Fabio Pellacini

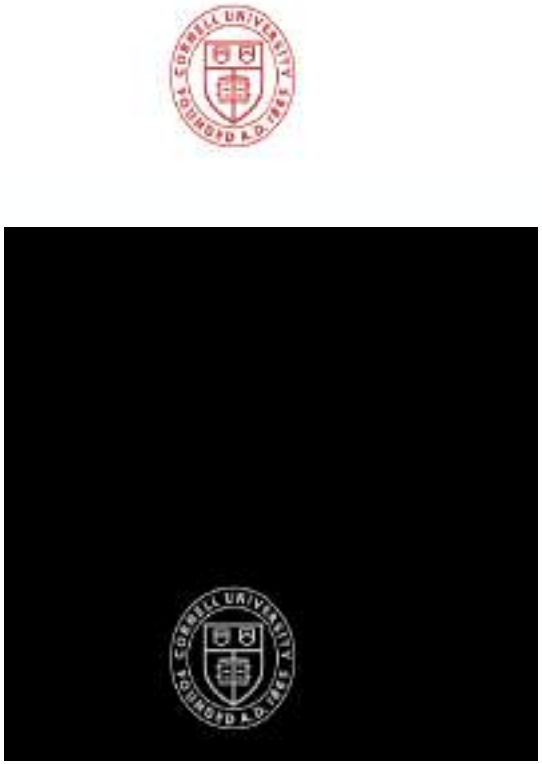
© 2020 Fabio Pellacini

© 2020 Fabio Pellacini

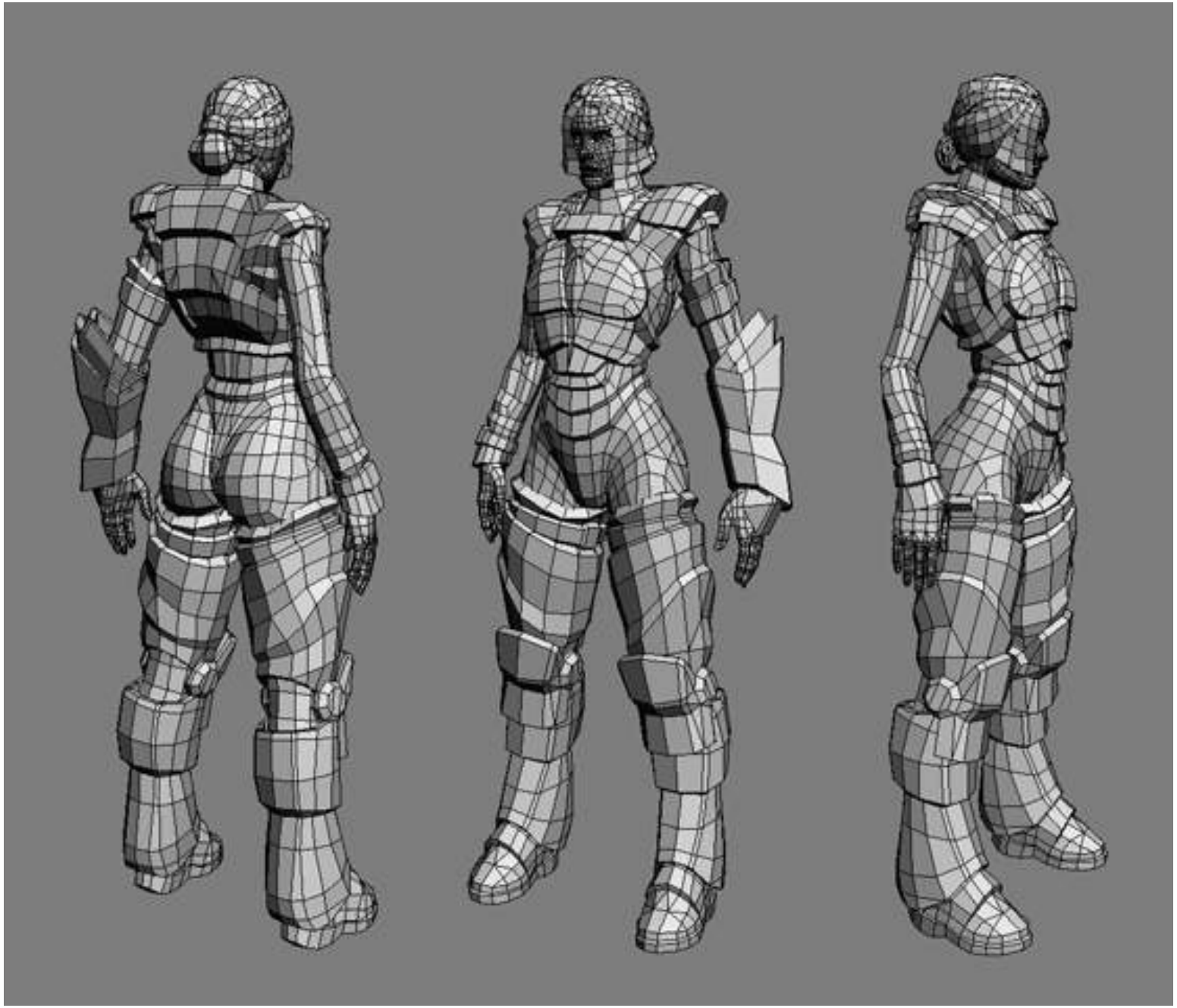


© 2020 Fabio Pellacini

© 2020 Fabio Pellacini



© 2020 Fabio Pellacini

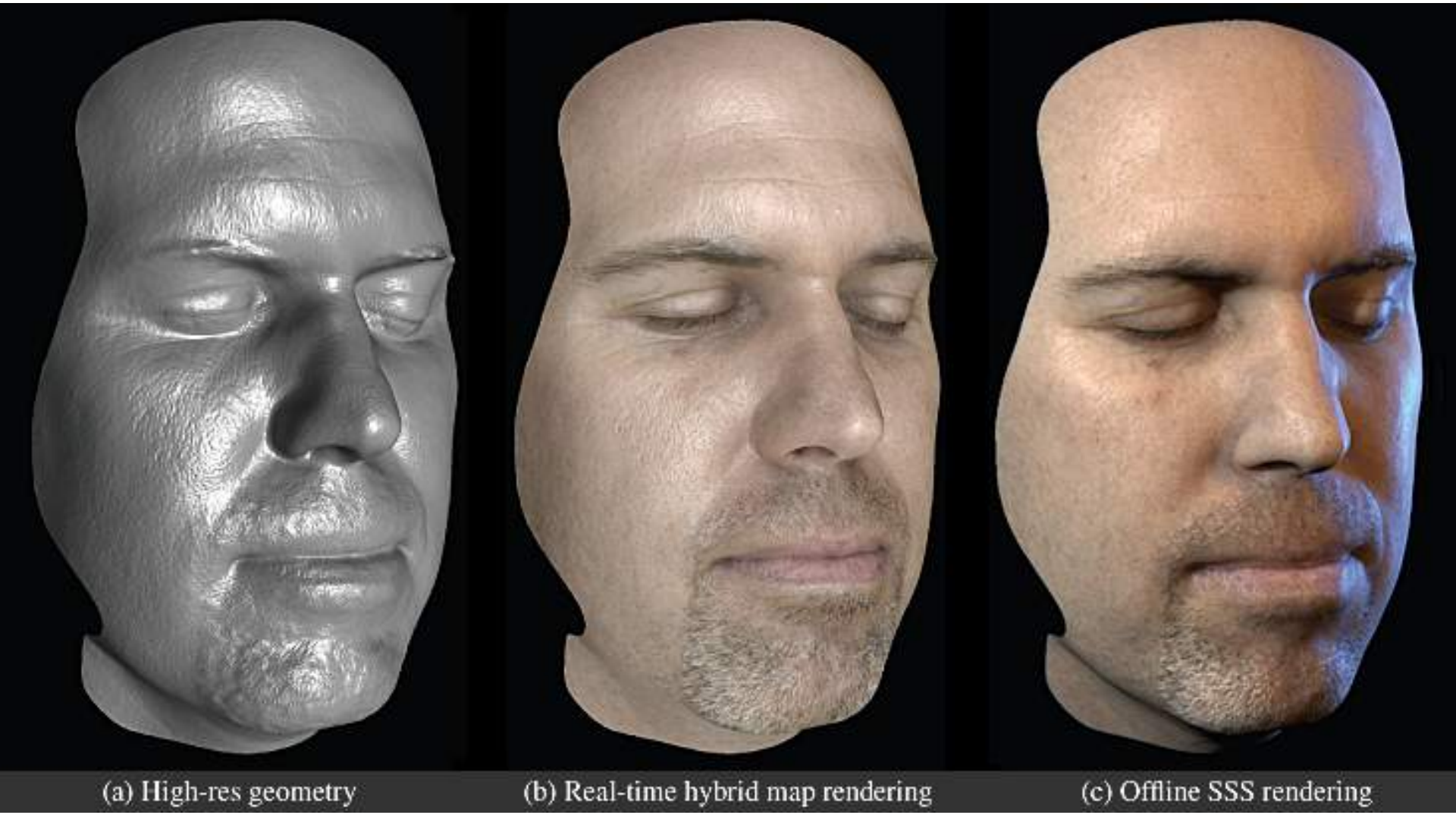


[source unknown]

© 2020 Fabio Pellacini •

[source unknown]

© 2020 Fabio Pellacini •

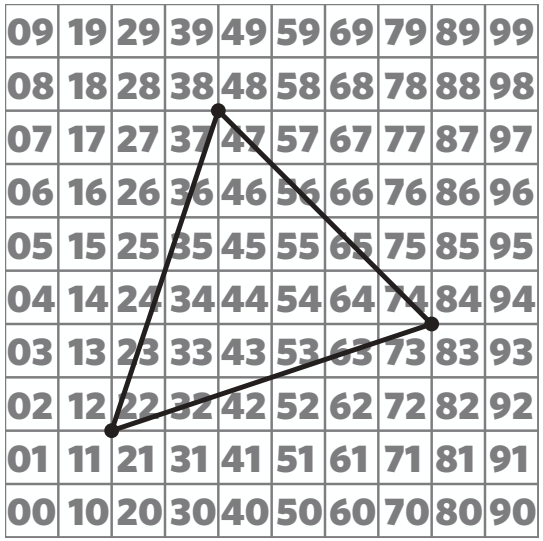


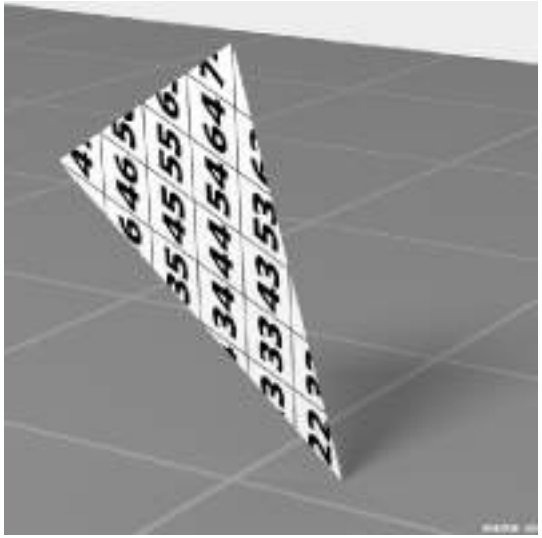
(a) High-res geometry (b) Real-time hybrid map rendering (c) Offline SSS rendering

[Paul Debevec]

Texture coordinates

- Texture coordinates are per-vertex data like vertex positions
  - can think of them as a second position: each vertex has a position in 3D space and in 2D texture space
- How to come up with (*u*,*v*)s for points inside triangles?

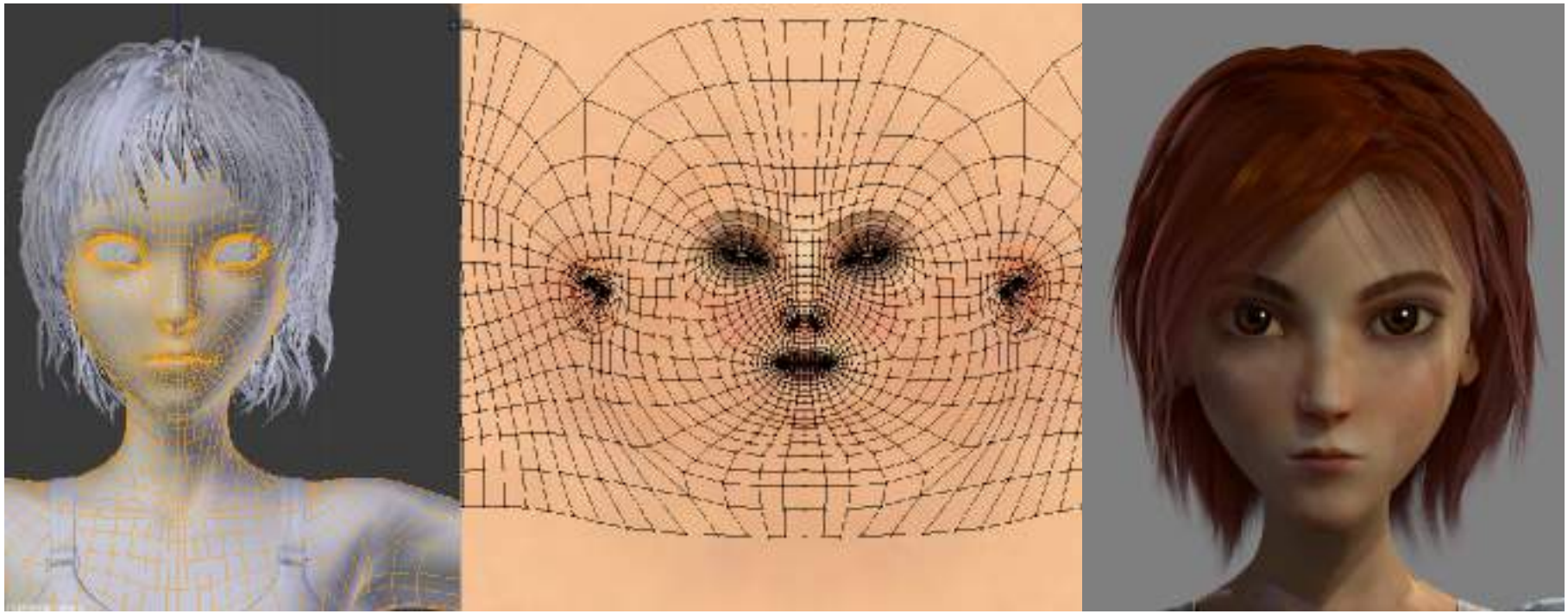




© 2020 Fabio Pellacini

Texture coordinates

- For each mesh vertex, assign a position in the image plane $[0,1]^2$
- Induces a mapping functions from 3D vertices to 2D pixels

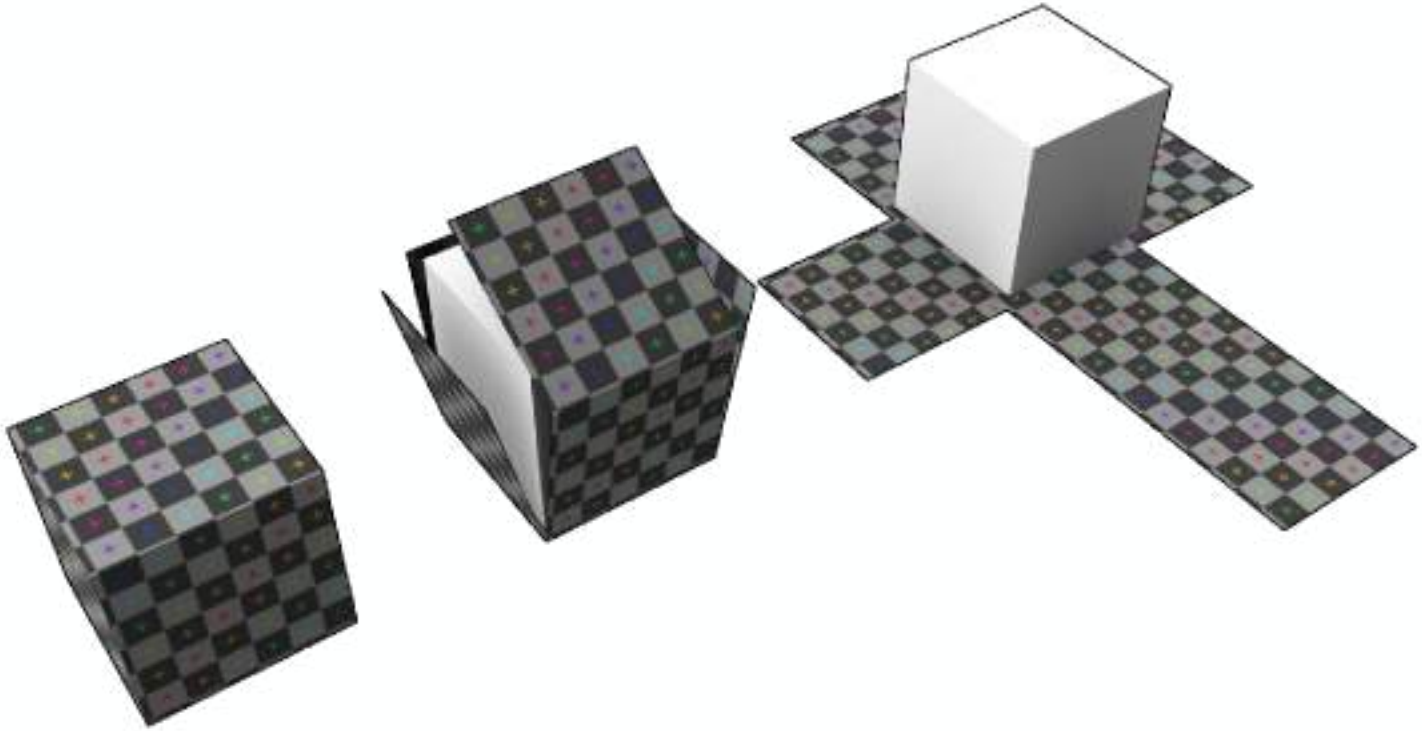


[Blender]

© 2020 Fabio Pellacini

Texture coordinates

- Assign this mapping is like “unrolling" the surface onto a plane
- Often done by hand, but some tools exist to help
- Will have discontinuities in most cases

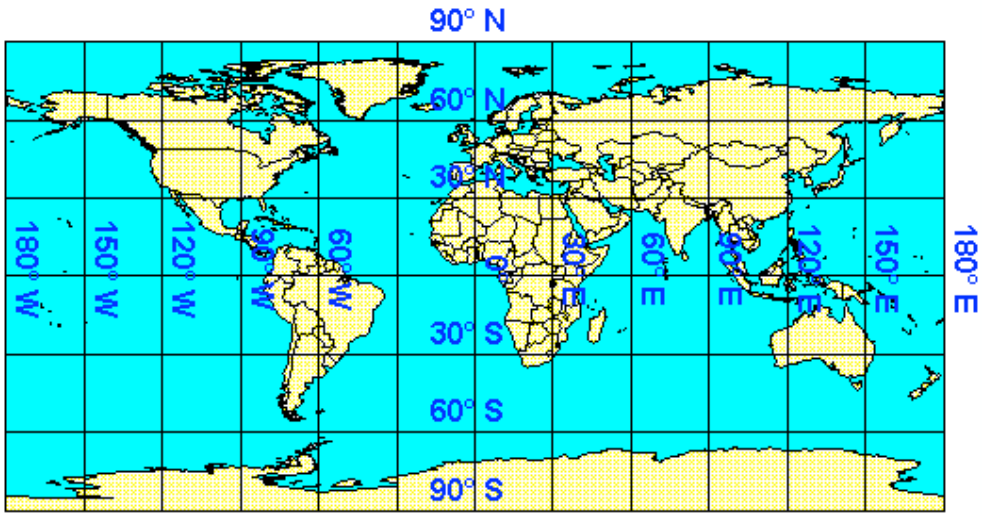


[Zepheris@WikimediaCommons]

© 2020 Fabio Pellacini

Texture coordinates

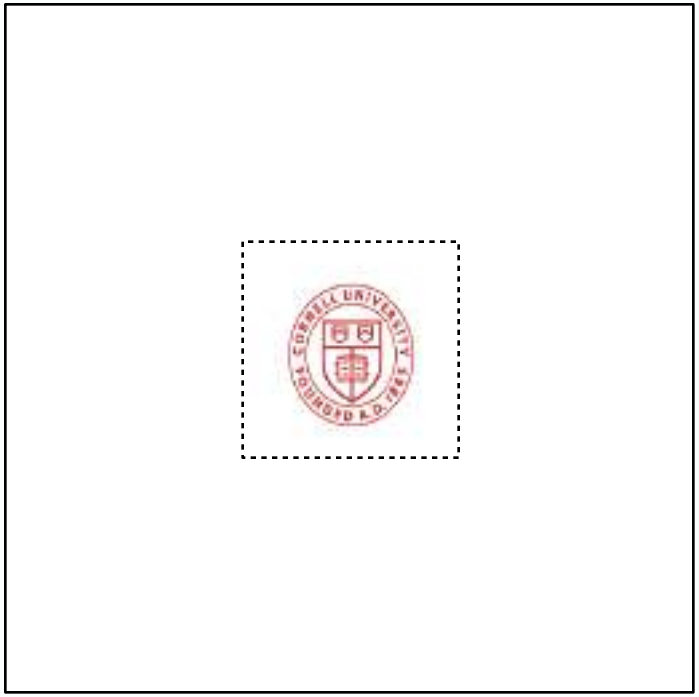
- Will also have distortions in most cases
  - Greenland looks really big in the map



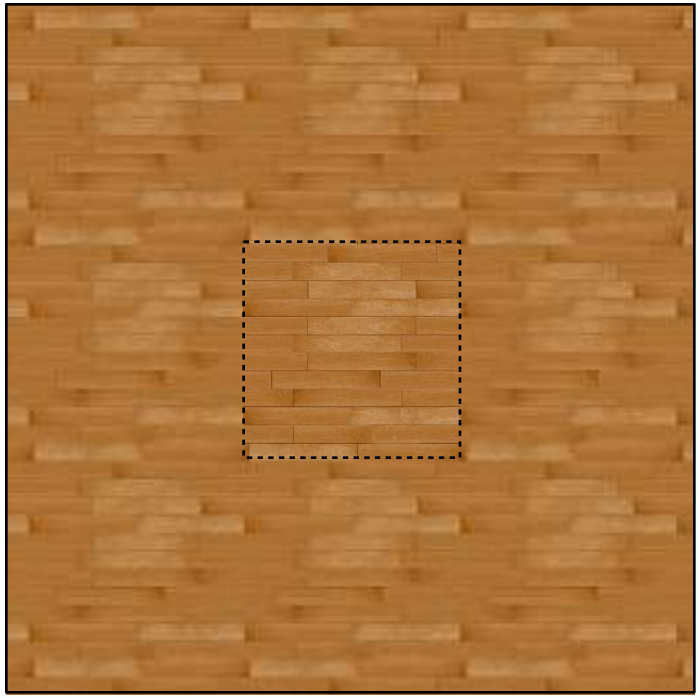
[map: Peter H. Dana]

© 2020 Fabio Pellacini

Wrapping modes



clamp



tiled

© 2020 Fabio Pellacini

Environment Maps

© 2017 Fabio Pellacini and Steve Marschner

Environment Maps

- Store far-away scene in an 360 panoramic image
- Main reasons: optimization and produces realistic rendering when used as a light source

[Paul Debevec]

© 2020 Fabio Pellacini

Environment Maps

- Store far-away scene in an 360 panoramic image
- Main reasons: optimization and produces realistic rendering when used as a light source

```
struct environment {
  frame3f frame;
  vec3f emission;          // emission
  texture* emission_tex;   // emission map
};
```


© 2020 Fabio Pellacini

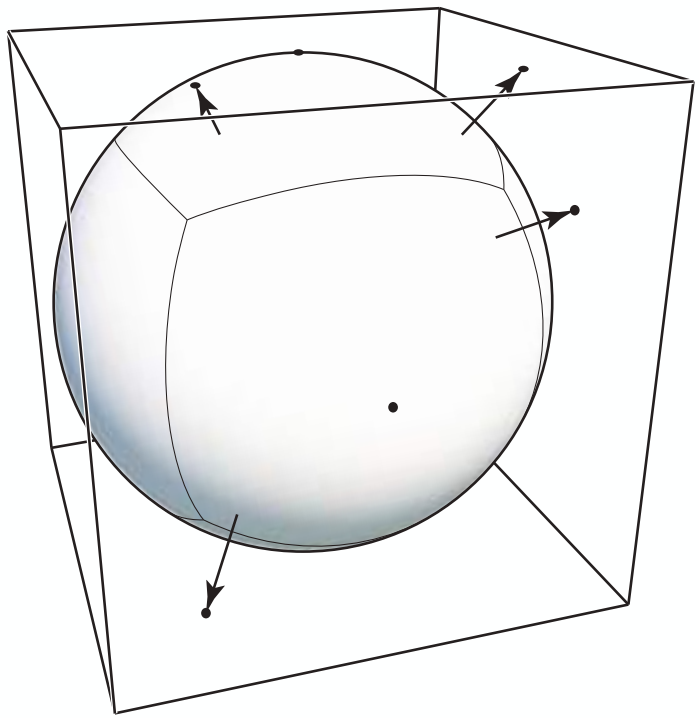
Lat-long Parametrization

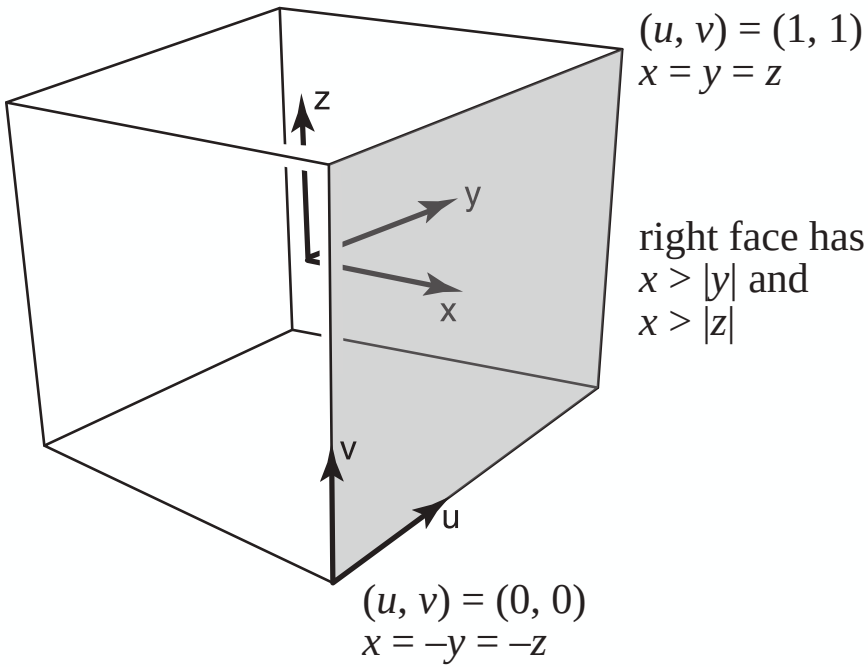
[GiantCowFilms]

© 2020 Fabio Pellacini

Cube Map

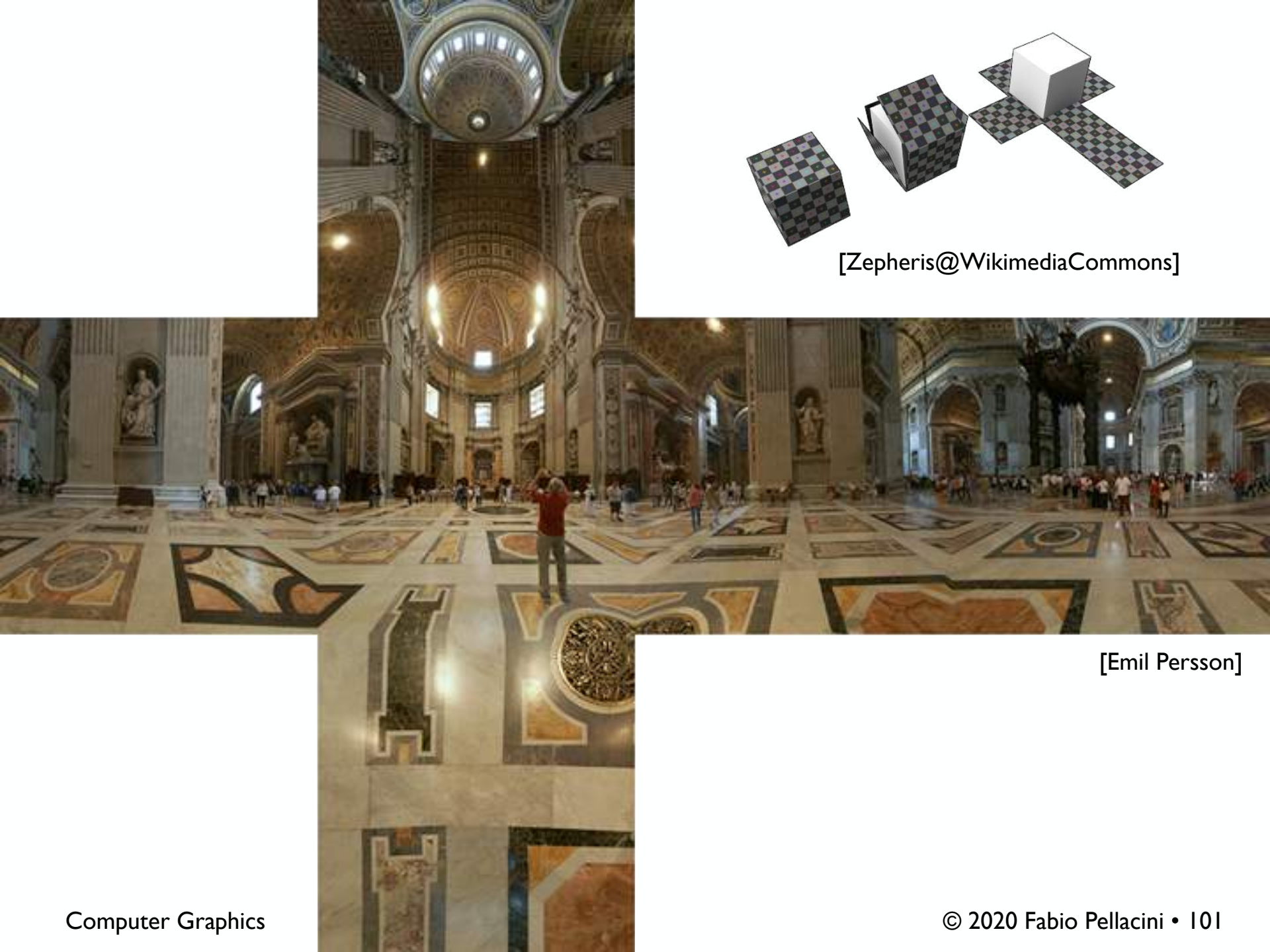
- Cube texture with six texture maps





© 2020 Fabio Pellacini

[Zepheris@WikimediaCommons]



[Emil Persson]

© 2020 Fabio Pellacini

HDRI illumination

[GiantCowFilms]

© 2020 Fabio Pellacini

HDRI illumination

[GiantCowFilms]

© 2020 Fabio Pellacini

Representing Scenes

© 2017 Fabio Pellacini and Steve Marschner

Representing Objects

- Need a way to associate geometry and materials
- Option 1: Add material inside a mesh
  - mixing geometry and shading
  - cannot re-used shape data
- Option 2: Bundle a mesh, a material and a frame into an object
  - with the coordinate frame, we can reposition the mesh
  - *instancing:* can represent large scenes by placing same objects

```
struct object {
  frame3f frame;      // coordinate frame
  shape*  shape;      // shape data
  material* material; // material
};
```


© 2020 Fabio Pellacini

Instancing

© 2020 Fabio Pellacini

Representing Scenes

- For now, we consider scenes as lists of objects, textures and materials and include a camera and environment map
- Later, we will introduce more complex scene modeling

```
struct scene {
  vector<camera*>      cameras;
  vector<object*>      objects;
  vector<shape*>       shapes;
  vector<material*>    materials;
  vector<texture*>     textures;
  vector<environment*> environments;
};
```


© 2020 Fabio Pellacini

Images and Display

© 2017 Fabio Pellacini and Steve Marschner

© 2017 Fabio Pellacini and Steve Marschner

What is an image?

- A photographic print
- A photographic negative?
- This projection screen
- Some numbers in RAM?

© 2017 Fabio Pellacini and Steve Marschner

An image is:

- A 2D distribution of intensity or color
- A function defined on a two-dimensional plane

$$I : \mathbb{R}^2 \rightarrow \ldots$$

- Note: no mention of pixels yet
- To do graphics, must:
  - represent images – encode them numerically
  - display images – realize them as actual intensity distributions

© 2017 Fabio Pellacini and Steve Marschner

Displays

© 2017 Fabio Pellacini and Steve Marschner

Display technologies

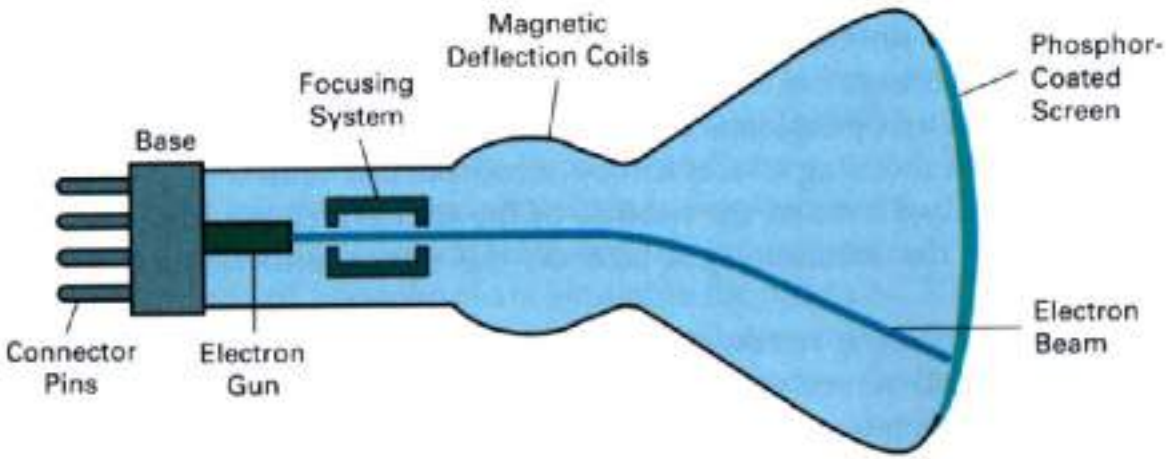
- Direct-view displays
  - Raster CRT display
  - LCD display
  - LED display

- Printers
  - Laser printer
  - Inkjet printer

© 2017 Fabio Pellacini and Steve Marschner

Cathode ray tube

- First widely used electronic display
  - developed for TV in the 1920s–1930s

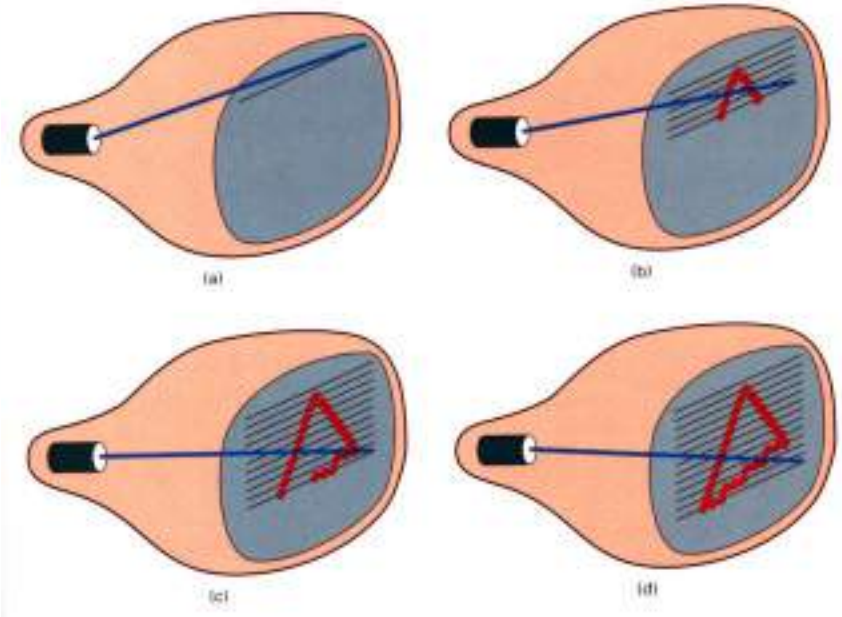


[H&B fig. 2-2]

© 2017 Fabio Pellacini and Steve Marschner

Raster CRT display

- Scan pattern fixed in display hardware
- Intensity modulated to produce image
- Originally for TV, continuous analog signal)
- For computer, intensity determined by contents of framebuffer

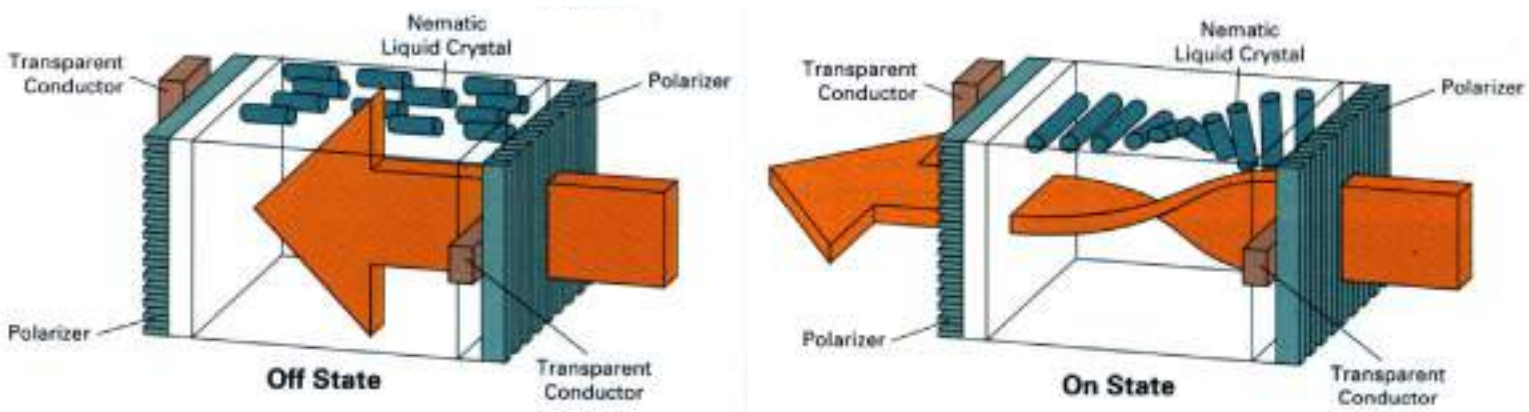


[H&B fig. 2-7]

© 2017 Fabio Pellacini and Steve Marschner

LCD flat panel display

- Principle: block or transmit light by twisting its polarization
- Illumination from backlight, either fluorescent or LED
- Intermediate intensity levels possible by partial twist
- Fundamentally raster technology



[H&B fig. 2-16]

© 2017 Fabio Pellacini and Steve Marschner

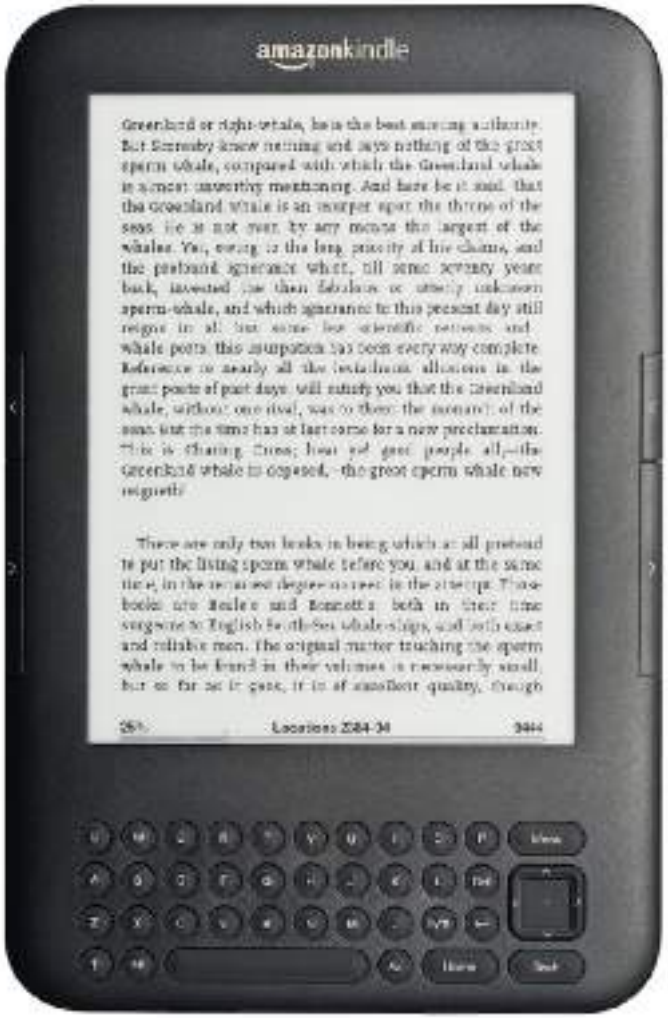
LED Displays

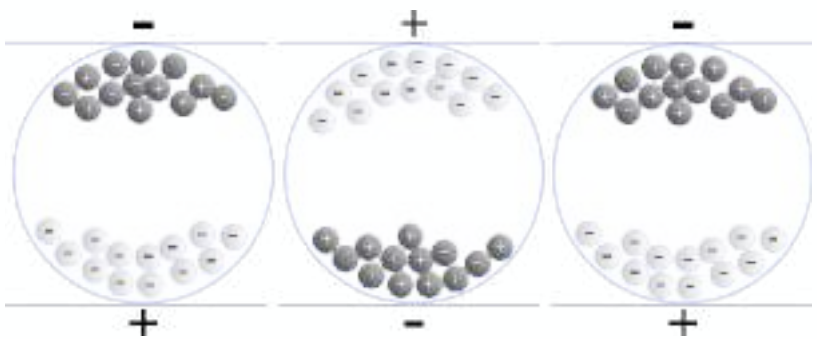
[Wikimedia Commons]

[Google—Nexus 4]

© 2017 Fabio Pellacini and Steve Marschner

Electrophoretic (electronic ink)

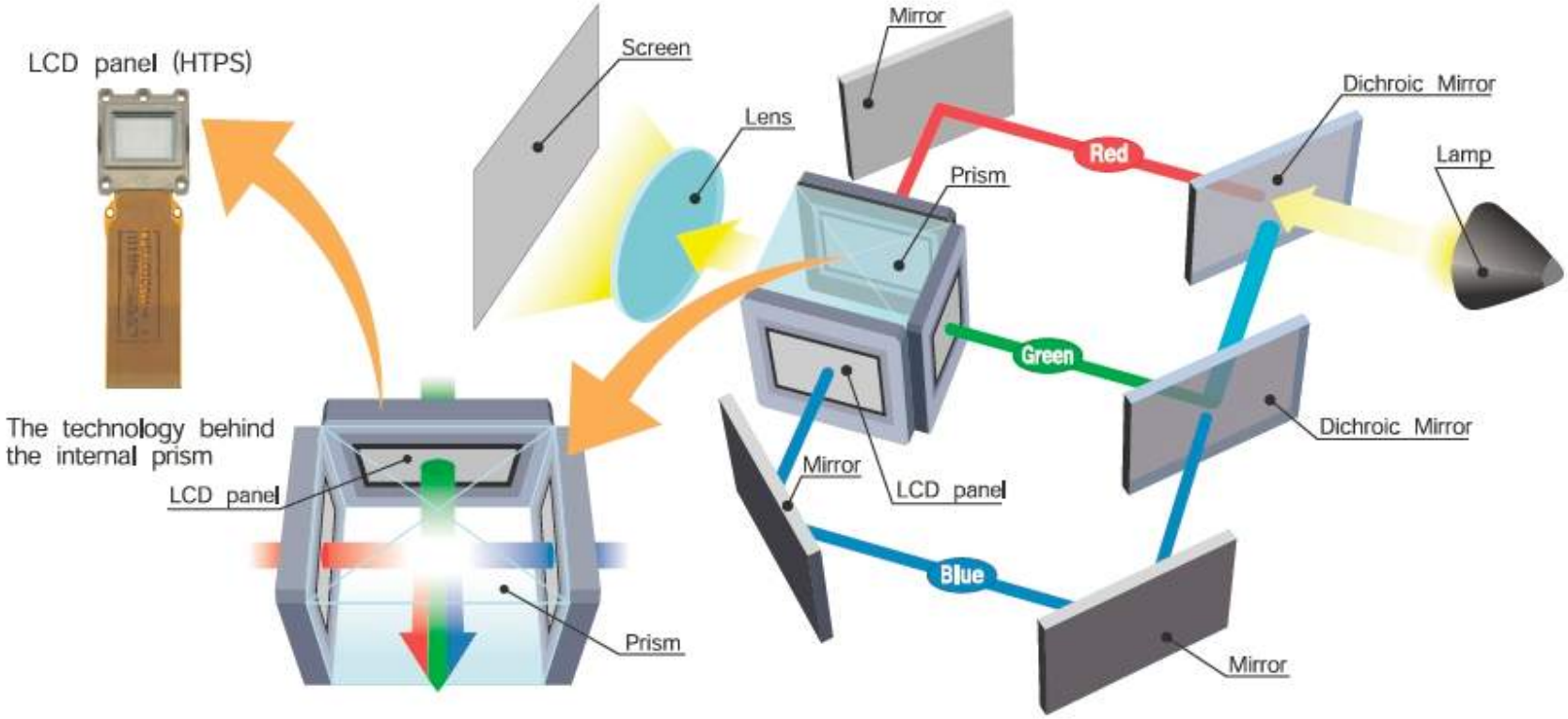




[Wikimedia Commons—Senarclens]

© 2017 Fabio Pellacini and Steve Marschner

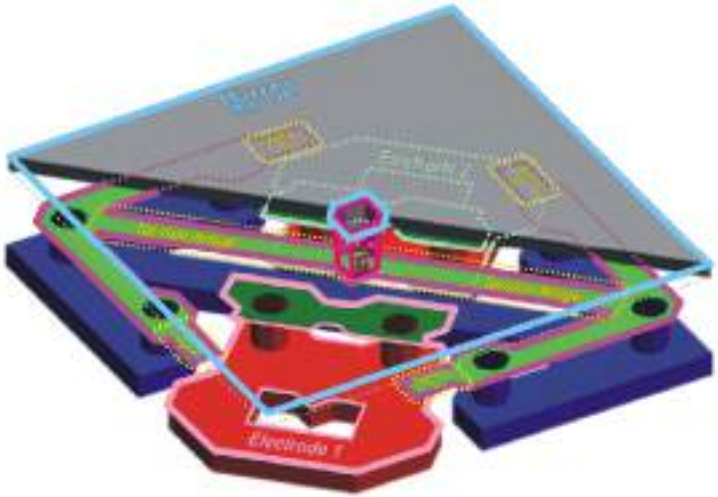
Projection displays: LCD

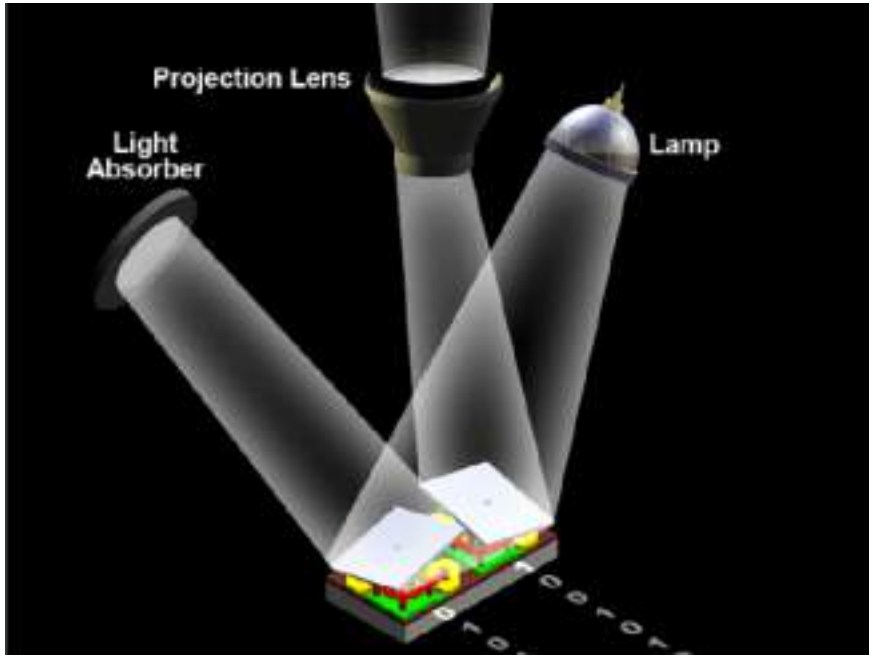


[Wikimedia Commons—Javachan]

© 2017 Fabio Pellacini and Steve Marschner

Projection displays: DLP



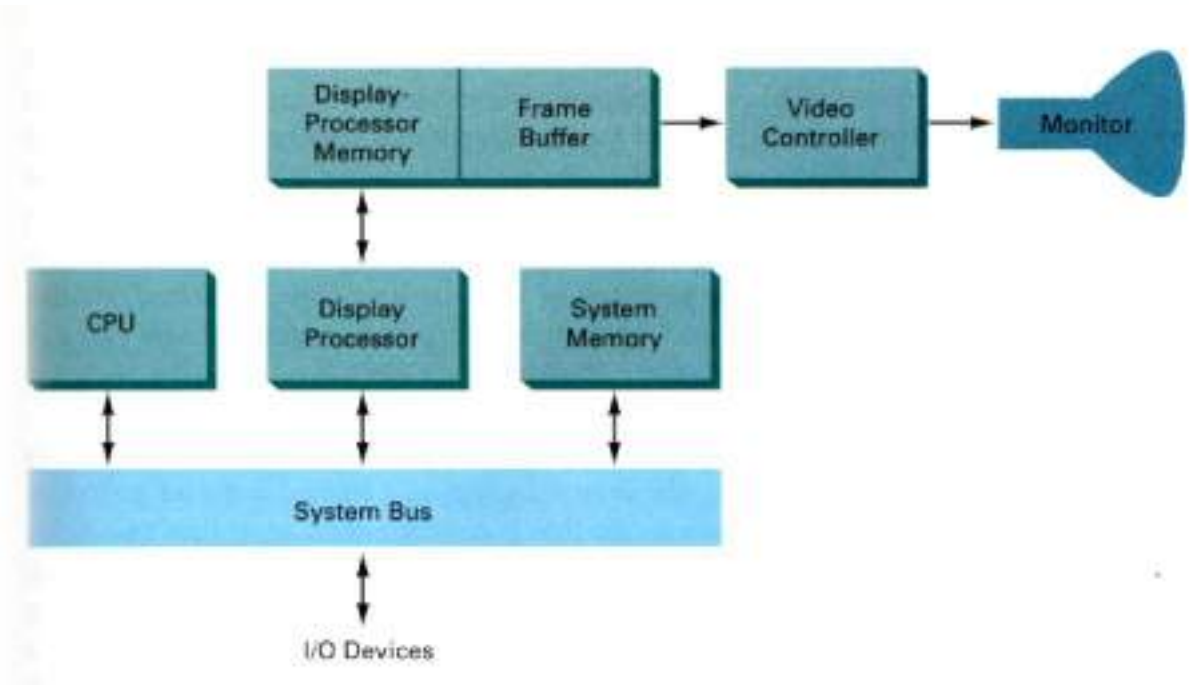


[Texas Instruments]

© 2017 Fabio Pellacini and Steve Marschner

Raster display system

- Screen image defined by a 2D array in RAM
- In most (but not all) systems today, it's in a separate memory from the normal CPU memory
- The memory area that maps to the screen is called the frame buffer

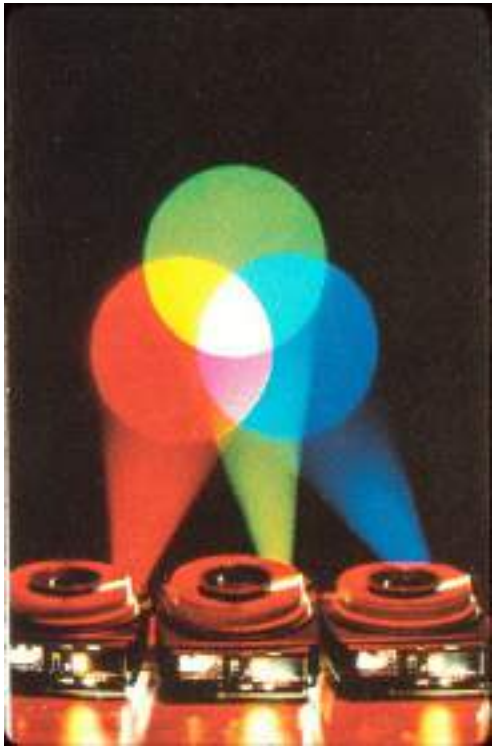


[H&B fig. 2-29]

© 2017 Fabio Pellacini and Steve Marschner

Color displays

- Operating principle: humans are trichromatic
  - match any color with blend of three
  - problem reduces to producing 3 images and blending

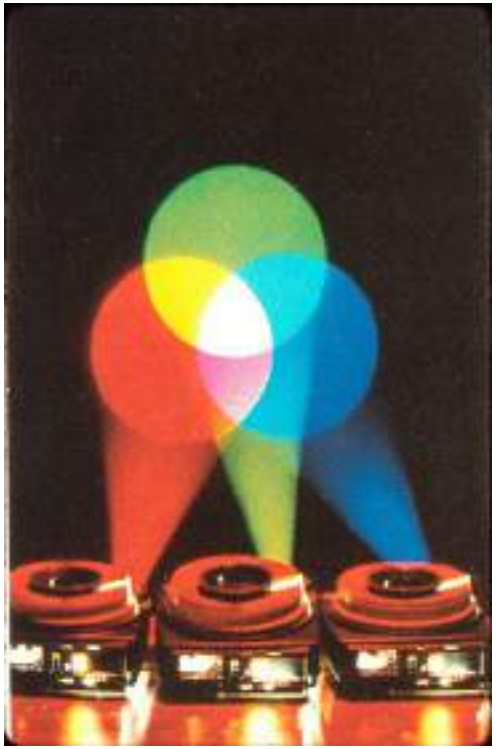


[cs417 S02 slides]

© 2017 Fabio Pellacini and Steve Marschner

Color displays

- Additive color: blend images by sum
  - e.g. overlapping projection, unresolved dots
  - R, G, B make good primaries

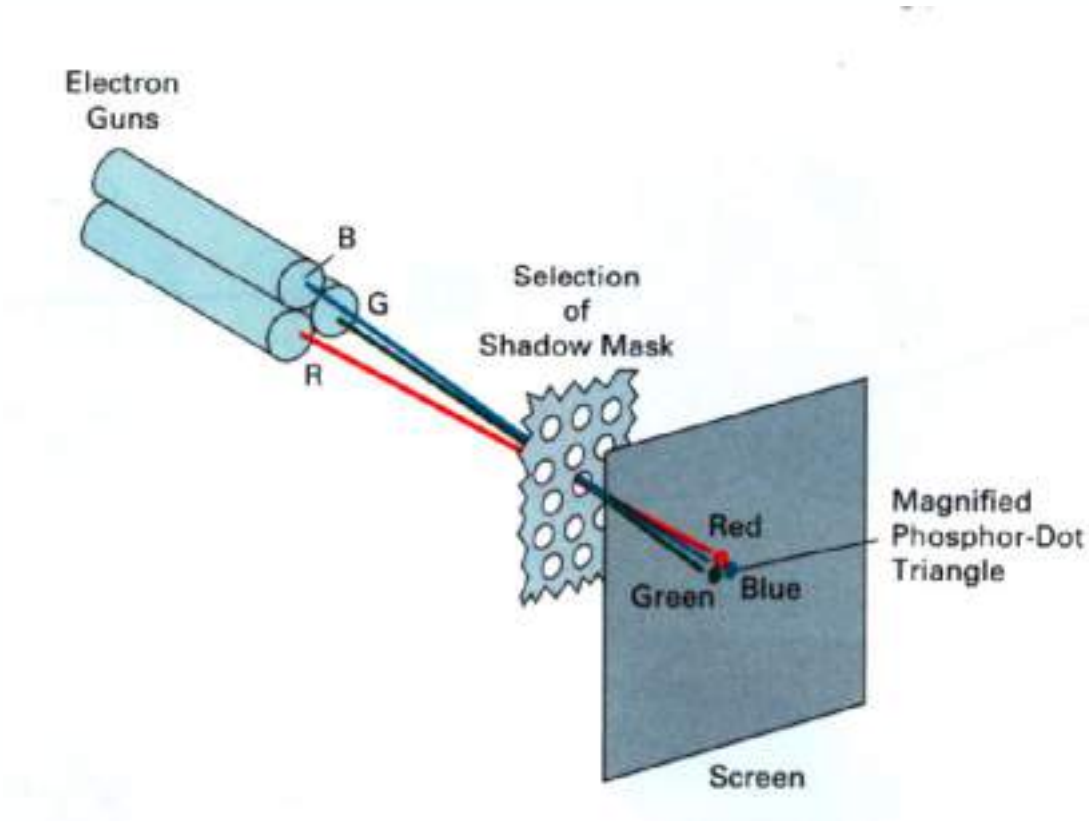


[cs417 S02 slides]

© 2017 Fabio Pellacini and Steve Marschner

Color displays

- CRT: phosphor dot pattern to produce finely interleaved color images

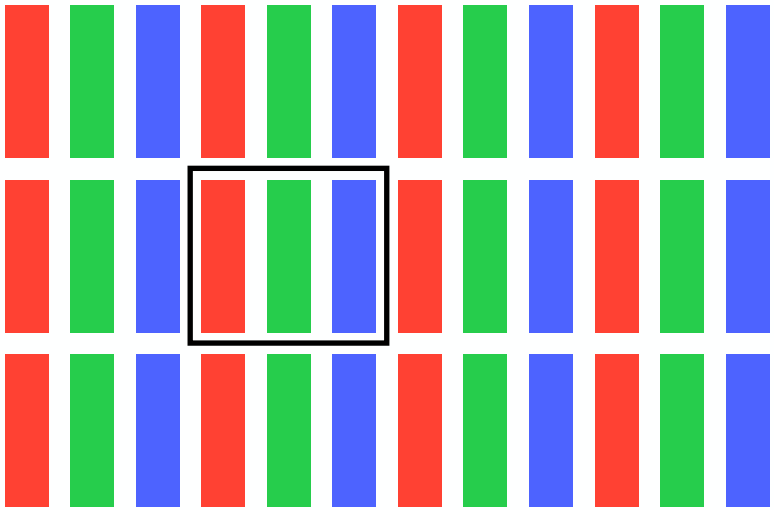


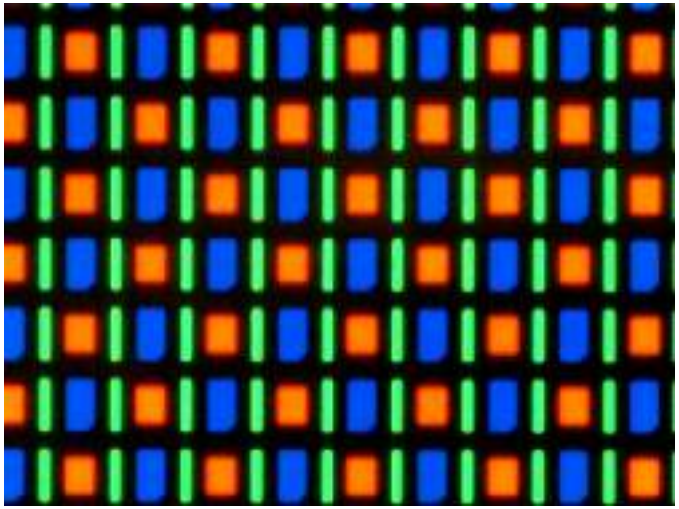
[H&B fig. 2-10]

© 2017 Fabio Pellacini and Steve Marschner

Color displays

- LCD, LED: interleaved R,G,B pixels



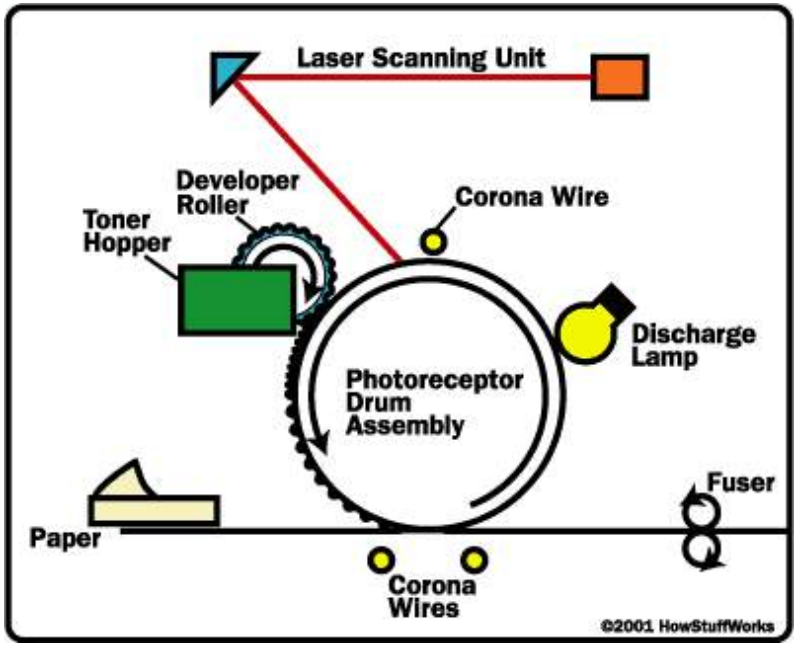


[Wikimedia Commons]

© 2017 Fabio Pellacini and Steve Marschner

Laser printer

- Xerographic process
- Like a photocopier but with laser-scanned raster as source image
- Key characteristics: image is binary, resolution is high, very small, isolated dots are not possible

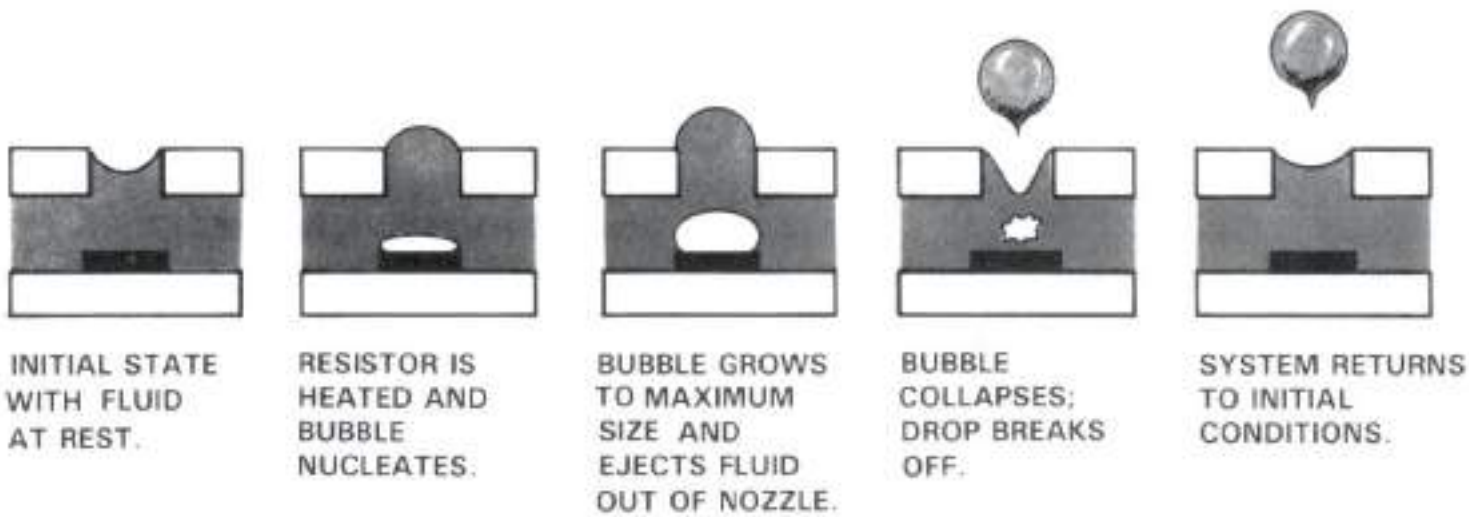


[howstuffworks.com]

© 2017 Fabio Pellacini and Steve Marschner

Inkjet printer

- Liquid ink sprayed in very small drops (picoliters)
- Head with many jets scans across paper
- Key characteristics: image is binary (no partial drops), isolated dots are reproduced well



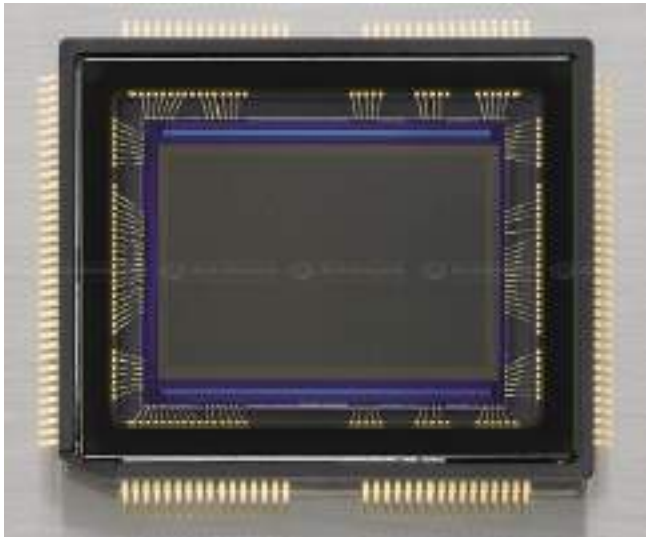
[cs417 S02 slides]

© 2017 Fabio Pellacini and Steve Marschner

Digital camera

- A raster input device
- Image sensor contains 2D array of photosensors

[PetaPixel.com]

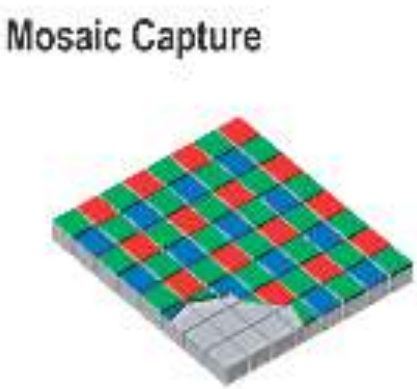


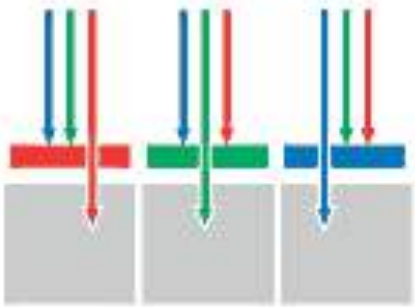
[dpreview.com]

© 2017 Fabio Pellacini and Steve Marschner

Digital camera

- Color typically captured using color mosaic





[Foveon]

© 2017 Fabio Pellacini and Steve Marschner

Image Representation

© 2017 Fabio Pellacini and Steve Marschner

Raster image representation

- All these devices suggest 2D arrays of values, called pixels
- Big advantage: represent arbitrary images
- Approximate arbitrary functions with increasing resolution

© 2017 Fabio Pellacini and Steve Marschner

Meaning of a raster image

- Meaning of a given array is a function on 2D
- Define meaning of array = result of output device?
  - that is, piecewise constant for LCD, blurry for CRT
  - but: we don't have just one output device
  - but: want to define images we can't display (e.g. too big)
- Abstracting from device, problem is reconstruction
  - image is a sampled representation
  - pixel means "this is the intensity around here"
    - LCD: intensity is constant over square regions
    - CRT: intensity varies smoothly across pixel grid

© 2017 Fabio Pellacini and Steve Marschner

Image Data Structure

- Option 1: Store images as 2D arrays
  - Not always supported by the programming language

```
struct image {
    DataType pixels[][];
}
```


© 2017 Fabio Pellacini and Steve Marschner

Image Data Structure

- Option 2: Store image contiguously, line-by-line
  - Most common representation
  - Access pixels manually

```
struct image {
  int width, height;
  DataType pixels[];

  DataType getPixel(int i, int j) {
    return pixels[j*width+i];
  }

  void setPixels(int i, int j, DataType v ) {
    pixels[j*width+i] = v;
  }
}
```


© 2017 Fabio Pellacini and Steve Marschner

Datatypes for raster images

- Bitmaps: boolean per pixel (1 bpp)
  - interp. = black and white; e.g. fax

$$I : \mathbb{R}^2 \to \{0, 1\}$$

© 2017 Fabio Pellacini and Steve Marschner

Datatypes for raster images

- Grayscale: integer per pixel
  - interp. = shades of gray; e.g. black-and-white print
  - precision: usually byte (8 bpp); sometimes 16 bpp

$$I : \mathbb{R}^2 \to [0, 1]$$

© 2017 Fabio Pellacini and Steve Marschner

Datatypes for raster images

- Color: 3 integers per pixel
  - interp. = full range of displayable color; e.g. color print
  - precision: usually byte[3] (24 bpp), sometimes short[3]

$$I : \mathbb{R}^2 \to [0,1]^3$$

© 2017 Fabio Pellacini and Steve Marschner

Datatypes for raster images

- Color: can we use less then 8 bits?
  - interp. = visible color banding
  - precision: example 12 bpp

$$I : \mathbb{R}^2 \to [0,1]^3$$

© 2017 Fabio Pellacini and Steve Marschner

Datatypes for raster images

- Floating point: 3 floats per pixel
  - more abstract, because no output device has infinite range
  - provides high dynamic range (HDR)
  - represent real scenes independent of display
  - becoming the standard intermediate format in graphics processor
  - current HDR TVs: 10–12 bits per channel
  - will discuss them later

$$I : \mathbb{R}^2 \to \mathbb{R}^3_+$$

© 2017 Fabio Pellacini and Steve Marschner

Storage requirements

- 1024x1024 image (1 megapixel)
  - bitmap: 128KB
  - grayscale 8bpp: 1MB
  - grayscale 16bpp: 2MB
  - color 24bpp: 3MB
  - floating-point HDR color: 12MB

© 2017 Fabio Pellacini and Steve Marschner

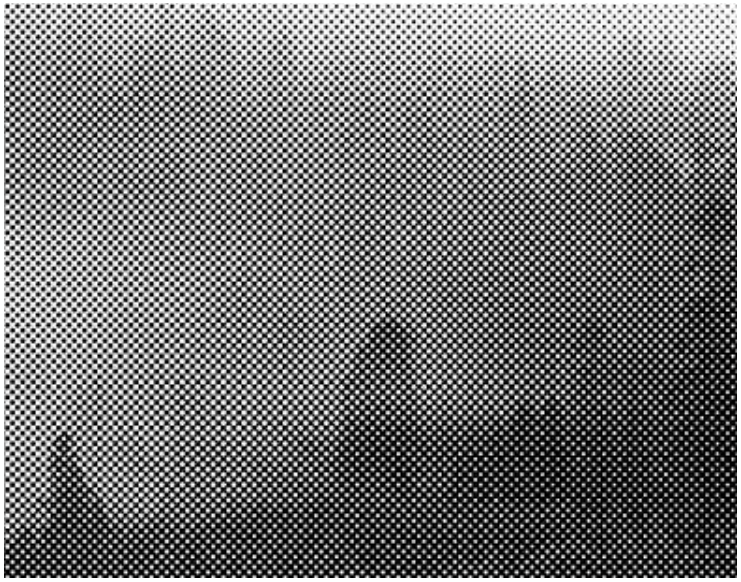
Dithering

- When decreasing bpp, we quantize
- Make choices consistently: banding
- Instead, be inconsistent—dither
  - turn on some pixels but not others in gray regions
  - a way of trading spatial for tonal resolution
  - choose pattern based on output device
  - laser, offset: clumped dots required (halftone)
  - inkjet, screen: dispersed dots can be used

© 2017 Fabio Pellacini and Steve Marschner

Diffusion Dither

- Produces regular grid of compact dots
  - based on traditional, optically produced halftones
  - produces larger dots

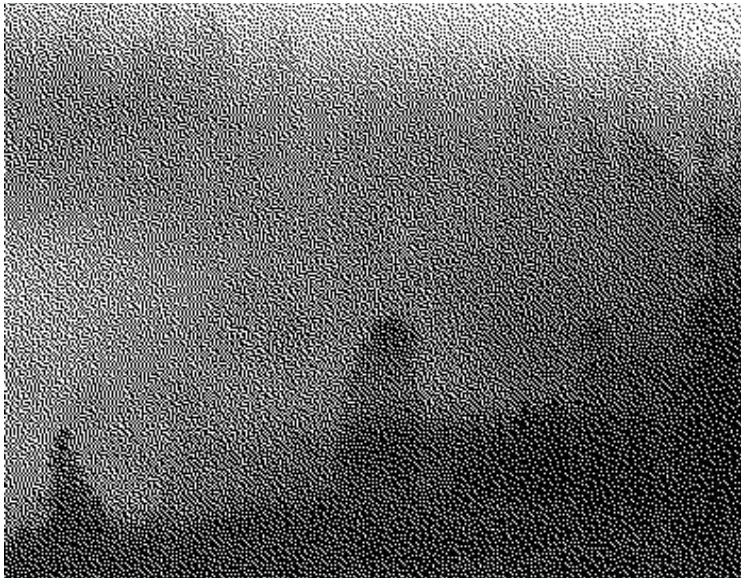


[photo: Philip Greenspun]

© 2017 Fabio Pellacini and Steve Marschner

Ordered Dither

- Produces scattered dots with the right local density
  - takes advantage of devices that can reproduce isolated dots
  - the modern winner for desktop printing



[photo: Philip Greenspun]

© 2017 Fabio Pellacini and Steve Marschner

Intensity encoding in images

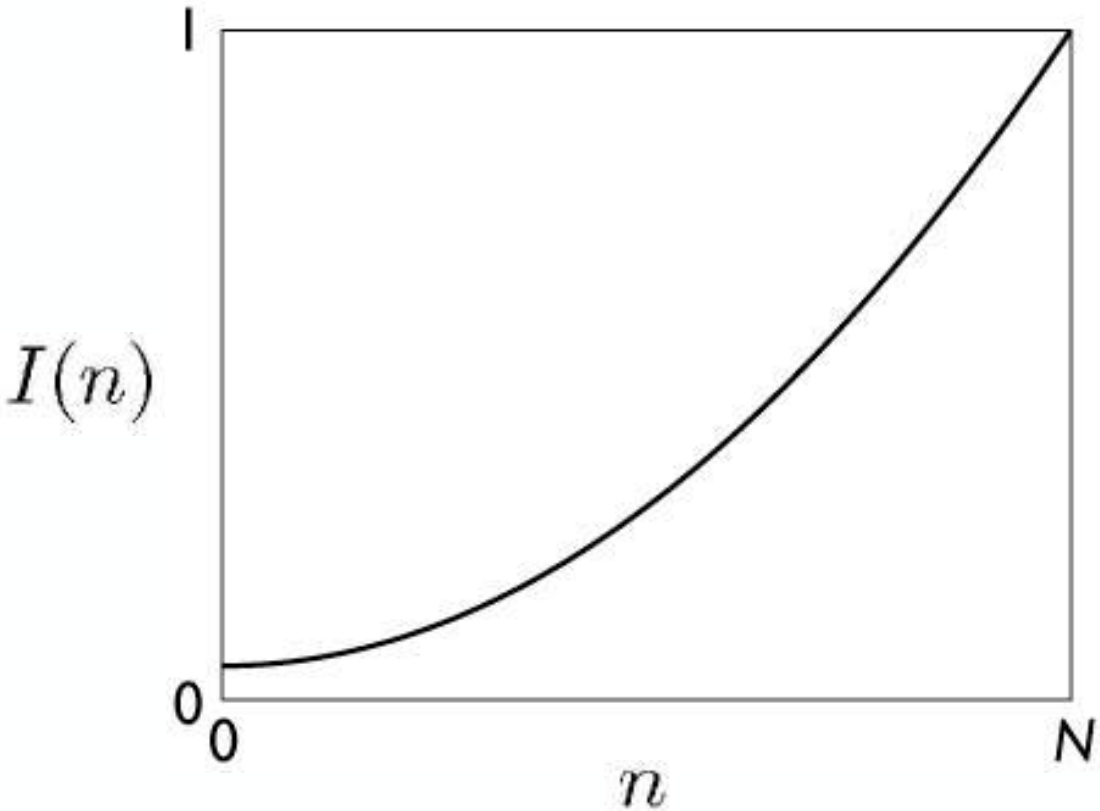
- What do the numbers in images (pixel values) mean?
- They determine how bright that pixel is
  - bigger numbers are brighter
- In HDR images (float), they are linearly related to the intensity
- In LDR images (integers), this mapping is not direct
  - transfer function *f*: function that maps input pixel values *n* to output luminance *I* of displayed image
  - determined by physical constraints of device and desired visual characteristics

$$I = f(n) \quad f : [0, N] \rightarrow [I_{min}, I_{max}]$$

© 2017 Fabio Pellacini and Steve Marschner

Typical Transfer Function

$$I = f(n) \quad f : [0, N] \rightarrow [I_{min}, I_{max}]$$



© 2017 Fabio Pellacini and Steve Marschner

Transfer function limits

- Maximum displayable intensity, Imax
  - how much power can be channeled into a pixel?
  - LCD: backlight intensity, transmission efficiency (<10%)
  - projector: lamp power, efficiency of imager and optics
- Minimum displayable intensity, Imin
  - light emitted by the display in its "off" state
  - CRT: stray electron flux,
  - LCD: polarizer quality

© 2017 Fabio Pellacini and Steve Marschner

Transfer function limits

- Viewing flare, k: light reflected by the display
  - very important factor determining image contrast in practice
  - 5% of Imax is typical in a normal office environment [sRGB spec]
  - much effort to make very black CRT and LCD screens
  - all-black decor in movie theaters

© 2017 Fabio Pellacini and Steve Marschner

Dynamic range

- Dynamic range: ratio between max and min display values
  - determines the degree of image contrast that can be achieved
  - a major factor in image quality

$$R_d = \frac{I_{max}}{I_{min}}$$

- Under non-ideal viewing condition, light is present in the environment, so dynamic range is reduced

$$R_d = \frac{I_{max} + I_{amb}}{I_{min} + I_{amb}}$$

© 2017 Fabio Pellacini and Steve Marschner

Dynamic range

- Ballpark values
  - Desktop display in typical conditions: 20:1
  - Photographic print: 30:1
  - Desktop display in good conditions: 100:1
  - High-end display under ideal conditions: 1000:1
  - Digital cinema projection: 1000:1
  - Photographic transparency (directly viewed): 1000:1
  - High dynamic range display: 10,000:1

© 2017 Fabio Pellacini and Steve Marschner

Transfer function shape

- Desirable property: the change from one pixel value to the next highest pixel value should not produce a visible contrast
  - otherwise smooth areas of images will show visible bands
- What contrasts are visible?
  - rule of thumb: under good conditions we can notice a 2% change in intensity
  - therefore we generally need smaller quantization steps in the darker tones than in the lighter tones
  - most efficient quantization is logarithmic

© 2017 Fabio Pellacini and Steve Marschner

How many levels are needed?

- 2% steps are most efficient
  - log 1.02 is about 1/120
  - 120 steps per decade of dynamic range

$$0 \mapsto I_{min}; 1 \mapsto 1.02 I_{min}; 2 \mapsto (1.02)^2 I_{min}; ...; n \mapsto (1.02)^n I_{min}$$

- Number of steps depends on dynamic range
  - 240 for desktop display
  - 360 to print to film
  - 480 to drive HDR display

© 2017 Fabio Pellacini and Steve Marschner

How many levels are needed?

- If we want to use linear quantization (equal steps)
  - one step must be < 2% (1/50) of lmin
  - 1500 for a print
  - 5000 for desktop display
  - 500,000 for HDR display
- Moral: 8 bits is just barely enough for low-end applications
  - but only if we are careful about quantization

© 2017 Fabio Pellacini and Steve Marschner

Intensity quantization

- Option 1: linear quantization
  - pro: simple, convenient, amenable to arithmetic
  - con: requires more steps (wastes memory)
  - need 12 bits for any useful purpose; more than 16 for HDR

$$I(n) = (n/N)I_{max}$$

- Option 2: power-law quantization
  - pro: fairly simple, approximates ideal exponential quantization
  - con: linearize before arithmetic, must agree on exponent
  - 8 bits are OK for many applications; 12 for more critical ones

$$I(n) = (n/N)^{\gamma} I_{max}$$

© 2017 Fabio Pellacini and Steve Marschner

Intensity quantization

- Option 3: floating-point quantization
  - floating points are also quantized to finite precision
  - pro: close to exponential; no parameters; arithmetic
  - con: takes 32 or 16 bits

$$I(x) = (x/w)I_{max}$$

© 2017 Fabio Pellacini and Steve Marschner

Gamma Correction

- Power-law quantization, or gamma correction is most popular
- Original reason: CRTs are like that
  - intensity on screen is proportional to (roughly) voltage^2
- Continuing reason: inertia + memory savings
  - inertia: gamma correction is close enough to logarithmic that there's no sense in changing
  - memory: gamma correction makes 8 bits per pixel an acceptable option

© 2017 Fabio Pellacini and Steve Marschner

Gamma correction

- We have computed intensities a that we want to display linearly
- In the case of an ideal monitor with zero black level and unit max

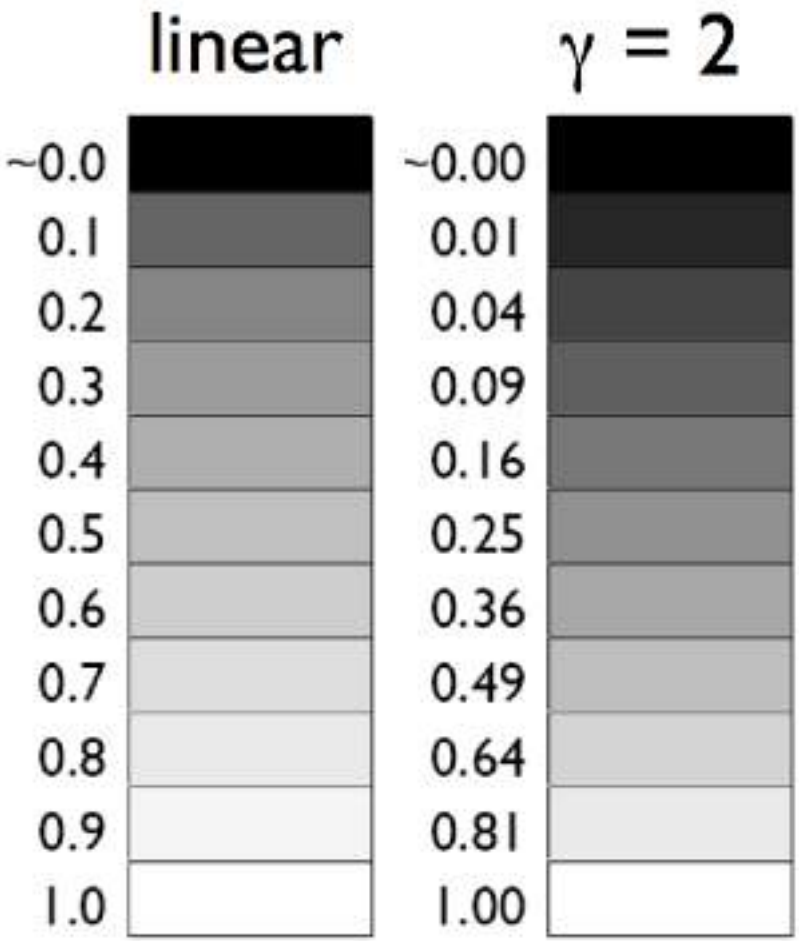
$$I(n) = (n/N)^\gamma$$

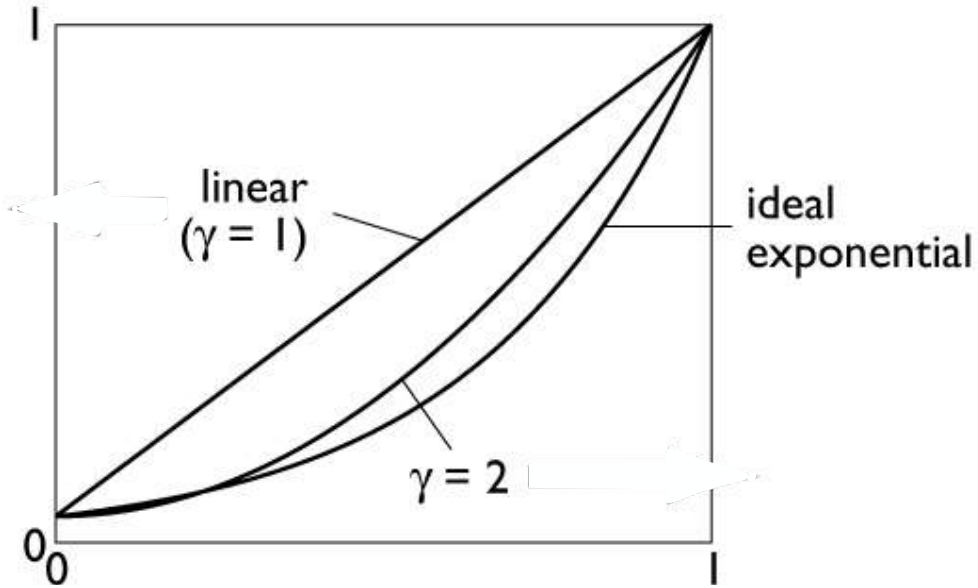
- This is the "gamma correction" recipe that has to be applied when computed values are converted to 8 bits for output
  - failing to do this (implicitly assuming gamma = 1) results in dark, oversaturated images
  - Typical value for gamma: 2.2

$$n(I) = NI^{\frac{1}{\gamma}}$$

© 2017 Fabio Pellacini and Steve Marschner

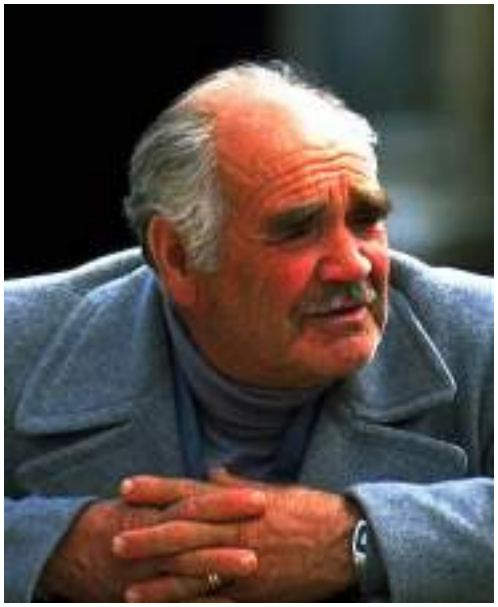
Gamma quantization



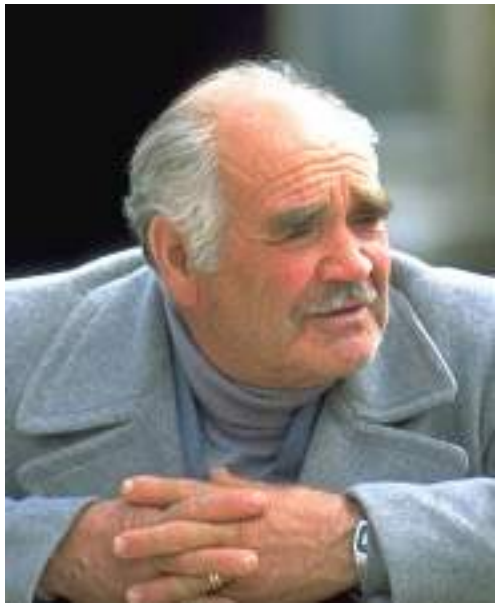


© 2017 Fabio Pellacini and Steve Marschner

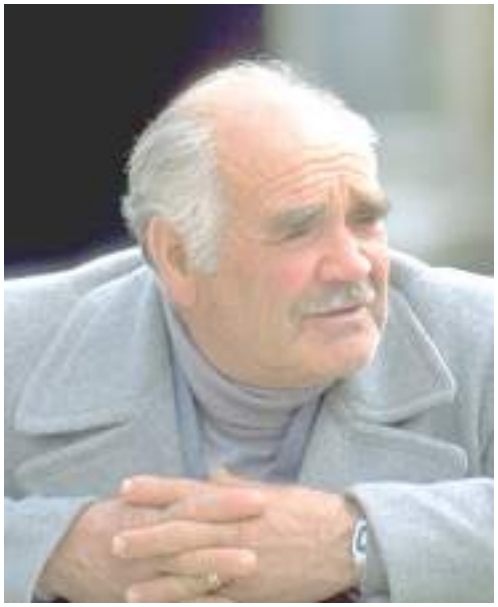
Gamma correction



γ lower than display



OK



γ higher than display

[Philip Greenspun]

© 2017 Fabio Pellacini and Steve Marschner

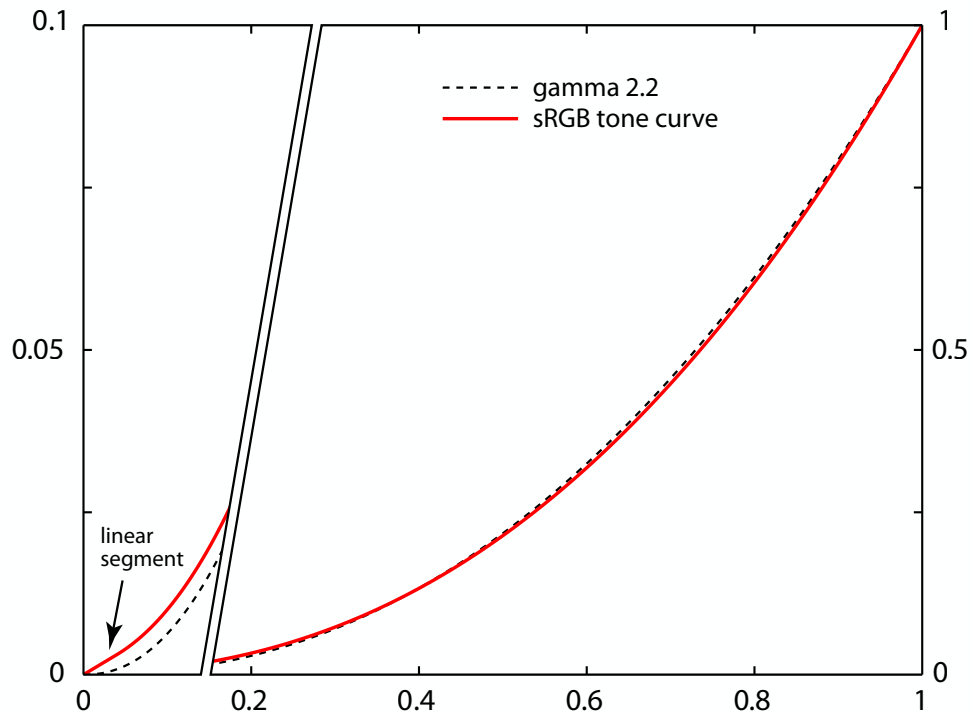
sRGB quantization curve

- The predominant standard for "casual color" in computer displays
  - backward compatible, monitors calibrated to sRGB by default
  - works well under imperfect conditions
  - approx. gamma 2.2

$$I(C) = \begin{cases} \frac{C}{12.92}, & C \leq 0.04045 \\ \left(\frac{C+a}{1+a}\right)^{2.4}, & C > 0.04045 \end{cases}$$

$$C = n/N$$

$$a = 0.055$$



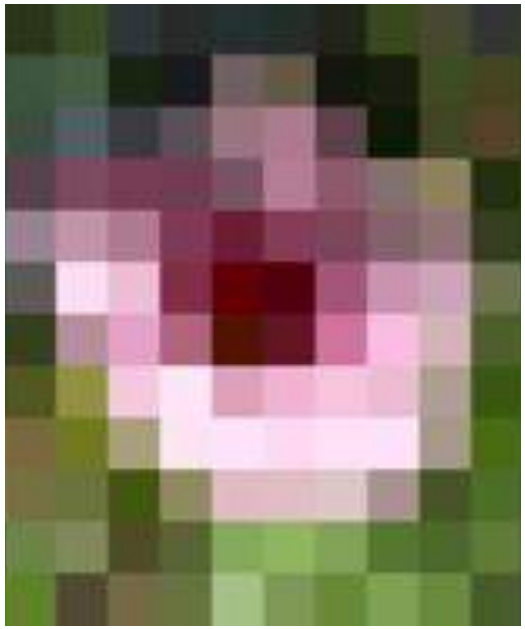
[derived from a figure by Dick Lyon]

© 2017 Fabio Pellacini and Steve Marschner

What is a pixel?

- Color "around" a point, not the pixel center
- Provide better approximation of the true values

Image



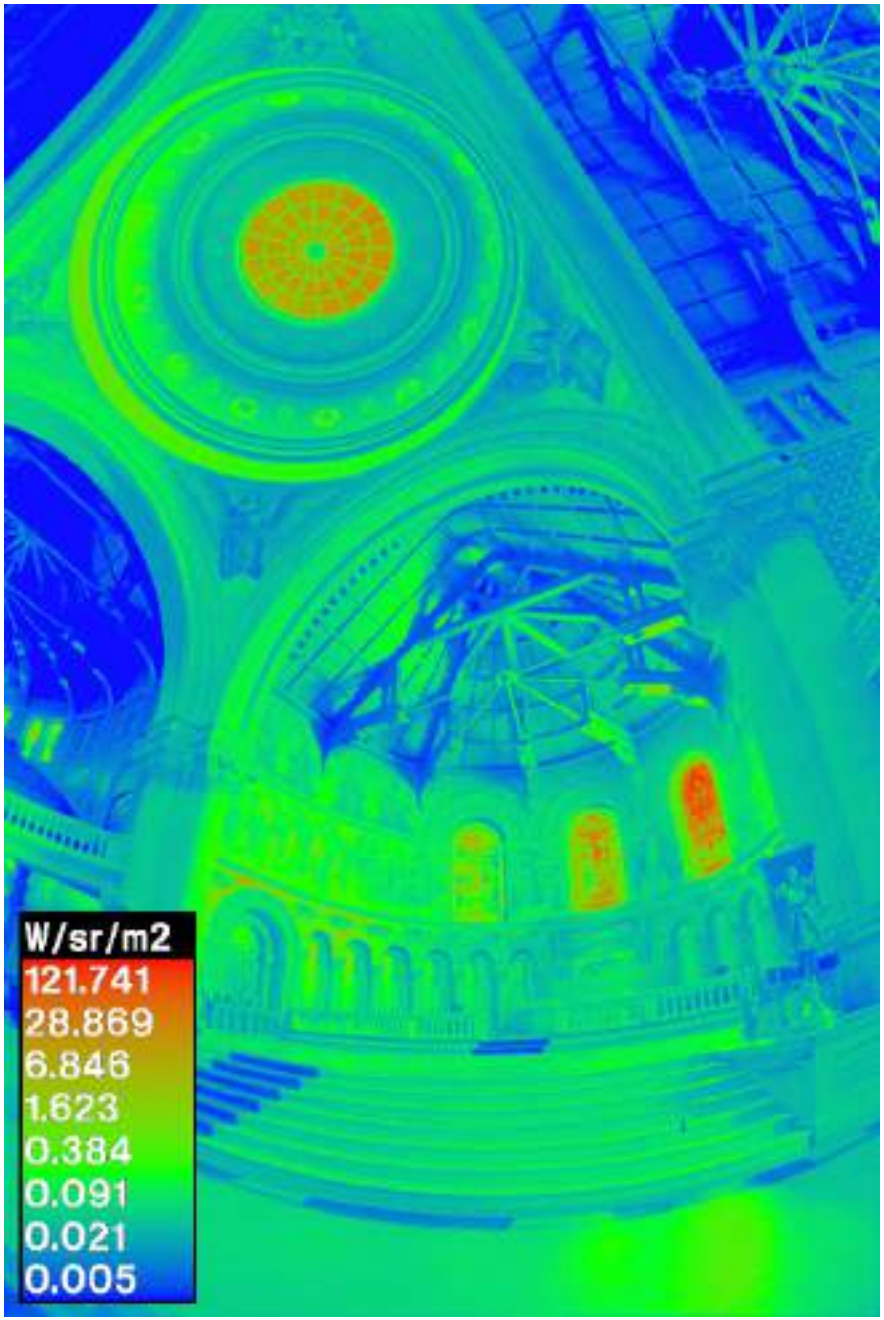
Area Average

Pixel Center

© 2017 Fabio Pellacini and Steve Marschner

HDR Images

© 2017 Fabio Pellacini and Steve Marschner

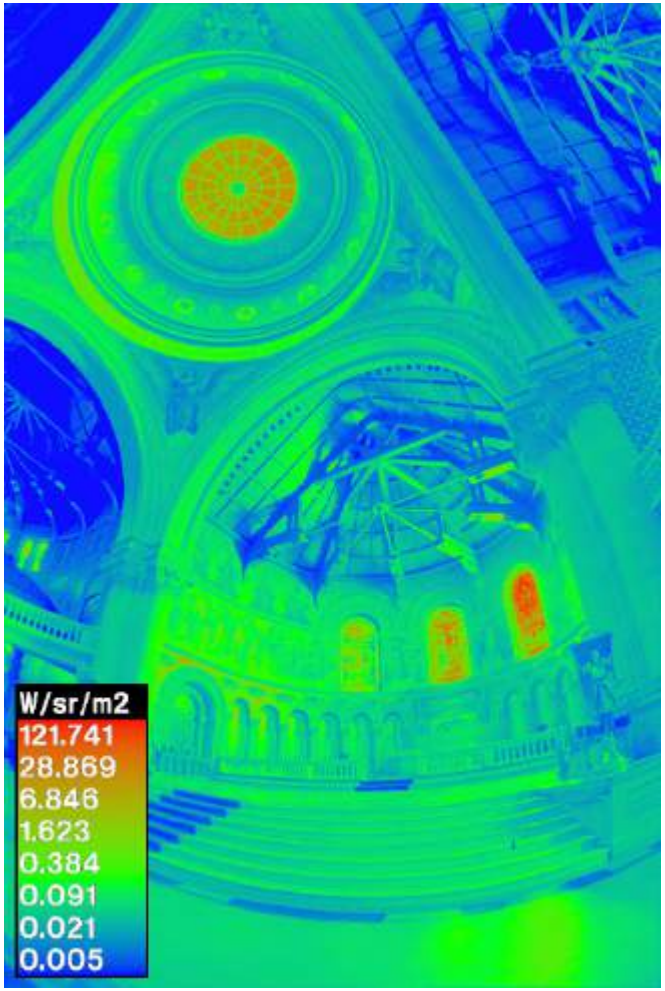


[Paul Debevec]

© 2017 Fabio Pellacini and Steve Marschner

HDR Images

- store illumination values directly
- values are linear and not clamped
  - no transfer function
  - requires floating point



[Paul Debevec]

© 2017 Fabio Pellacini and Steve Marschner

Capturing HDR Images

- capture multiple exposure, each of which is clamped
- aligned them so that pixels corresponds
- blend the "middle" portion of the range, to avoid clamped regions

© 2017 Fabio Pellacini and Steve Marschner

Viewing HDR images

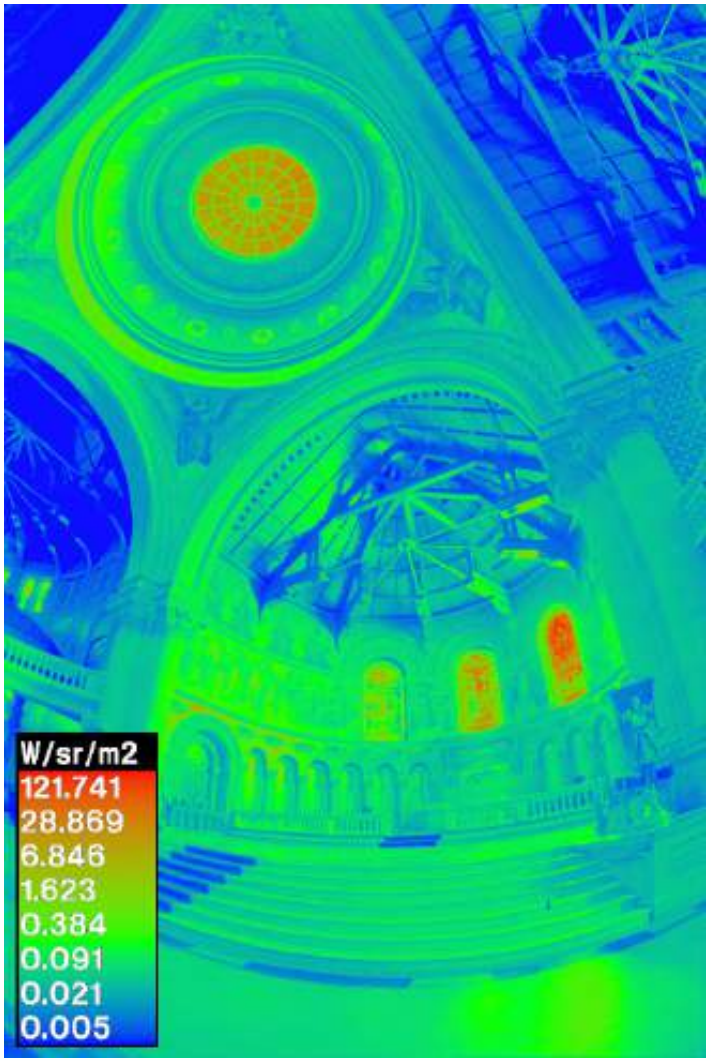
- HDR images store all illumination values
- But displays can only reproduce between a min and max value
- Tone mapping: Reduce HDR range to display range
- Simple method: scale HDR values, apply gamma, and clamp
  - scale is often expressed in power of twos called *exposure*

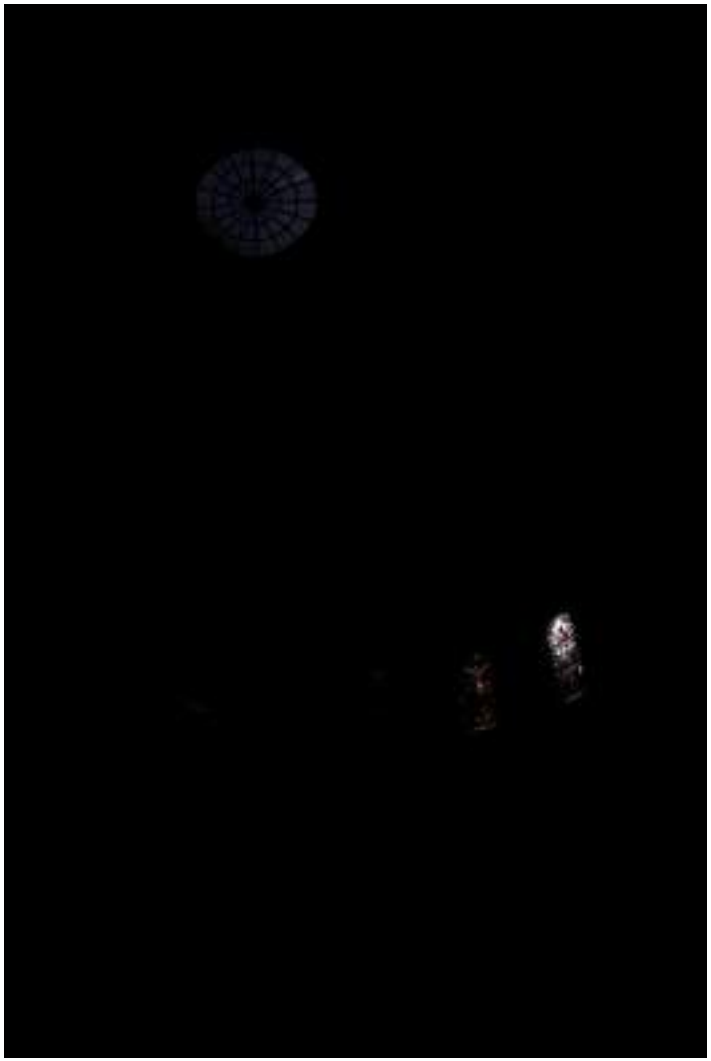
$$I_{LDR} = \min(1, (sI_{HDR})^{\gamma}) \quad s = 2^{exposure}$$

- For more artistic control: use different non-linear curve
  - control over mid-tone contrast and min/max values
- For more "correctness": simulate visual system

© 2017 Fabio Pellacini and Steve Marschner

Linear scale

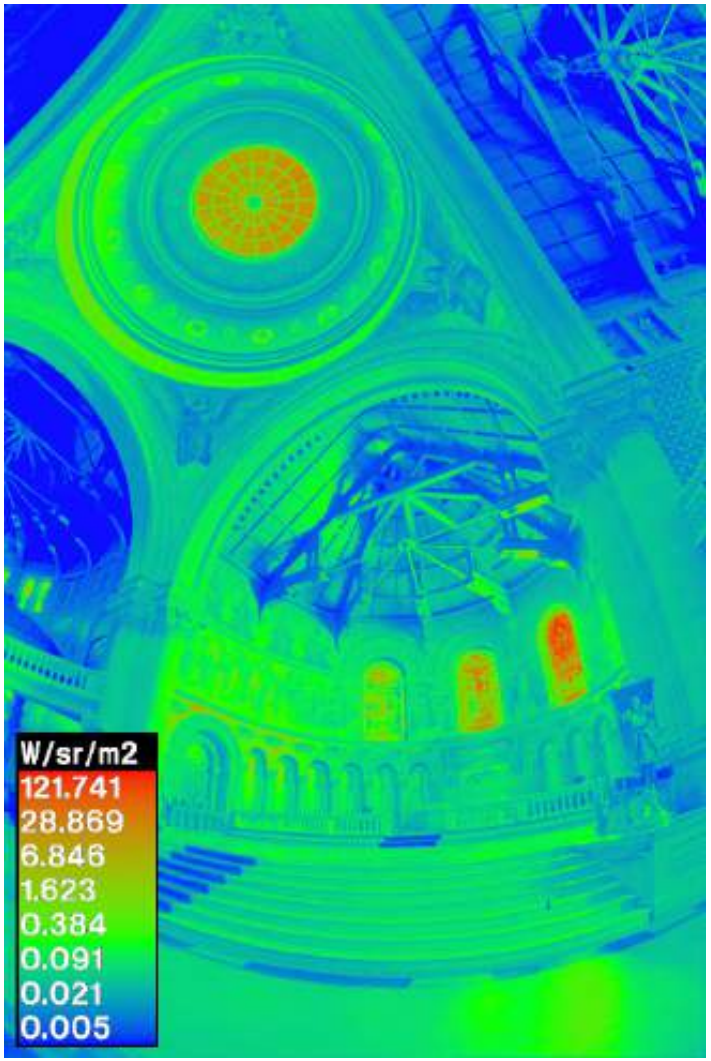




[Paul Debevec]

© 2017 Fabio Pellacini and Steve Marschner

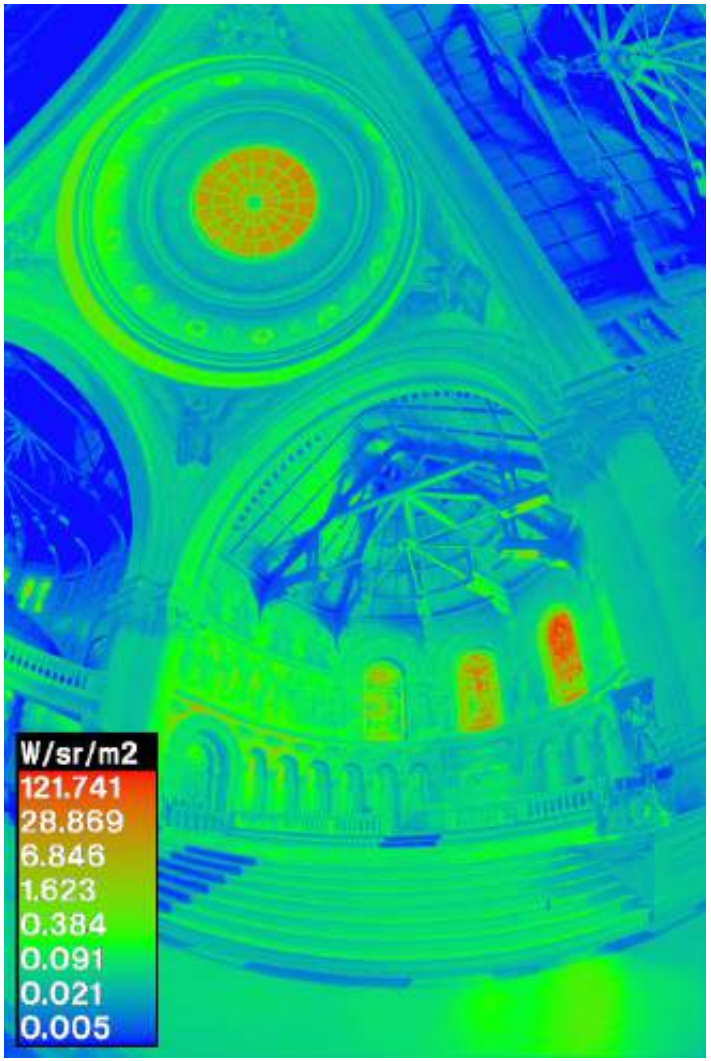
Select range via exposure



[Paul Debevec]

© 2017 Fabio Pellacini and Steve Marschner

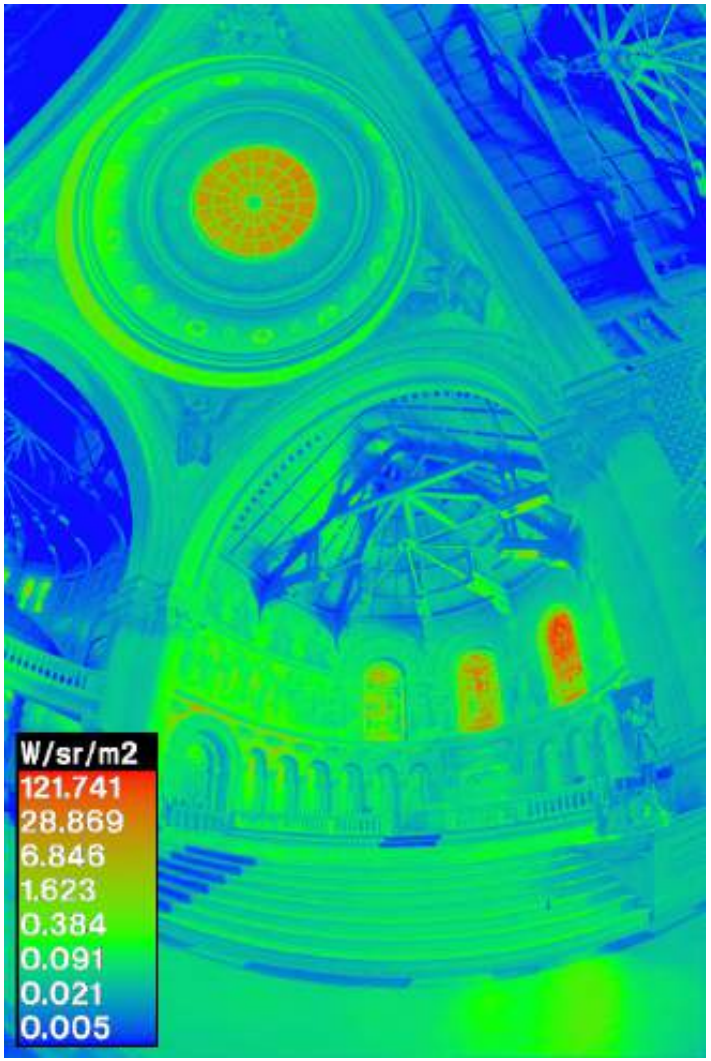
Non-linear correction (gamma)



[Paul Debevec]

© 2017 Fabio Pellacini and Steve Marschner

Simulate Visual System



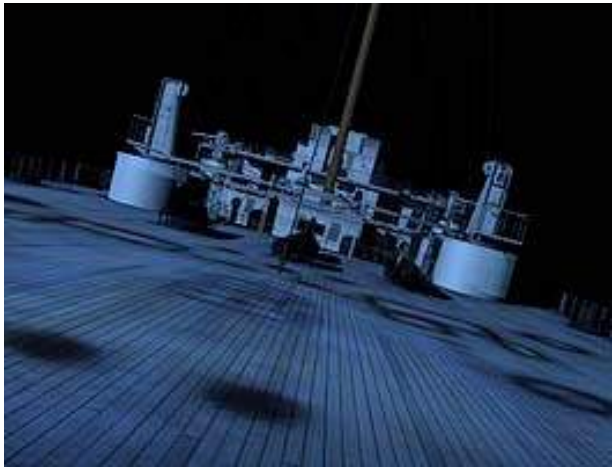
[Paul Debevec]

© 2017 Fabio Pellacini and Steve Marschner

Compositing

© 2017 Fabio Pellacini and Steve Marschner

Compositing



[Titanic ; DigitalDomain; vfxhq.com]

© 2017 Fabio Pellacini and Steve Marschner

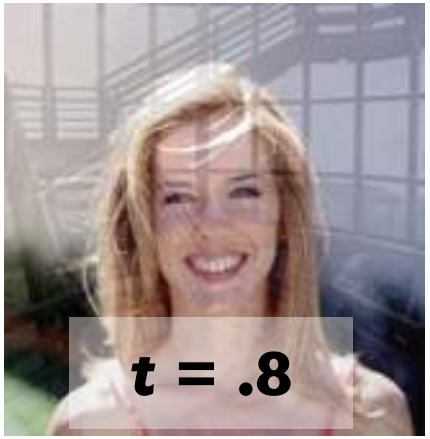
Combining images

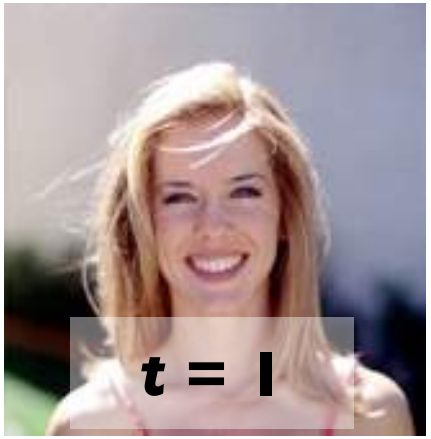
- Trivial example: video crossfade
  - smooth transition from one to another by *linear interpolation*
  - note that weights sum to 1.0
  - no brightening or darkening
  - no out-of-range values
- Written in vector notation as

$$\mathbf{c}_C = t\mathbf{c}_A + (1-t)\mathbf{c}_B \qquad \begin{bmatrix} r_C \\ g_C \\ b_C \end{bmatrix} = \begin{bmatrix} tr_A + (1-t)r_B \\ tr_A + (1-t)g_B \\ tr_A + (1-t)b_B \end{bmatrix}$$

© 2017 Fabio Pellacini and Steve Marschner

Combining images





[Chuang et al. / Corel]

© 2017 Fabio Pellacini and Steve Marschner

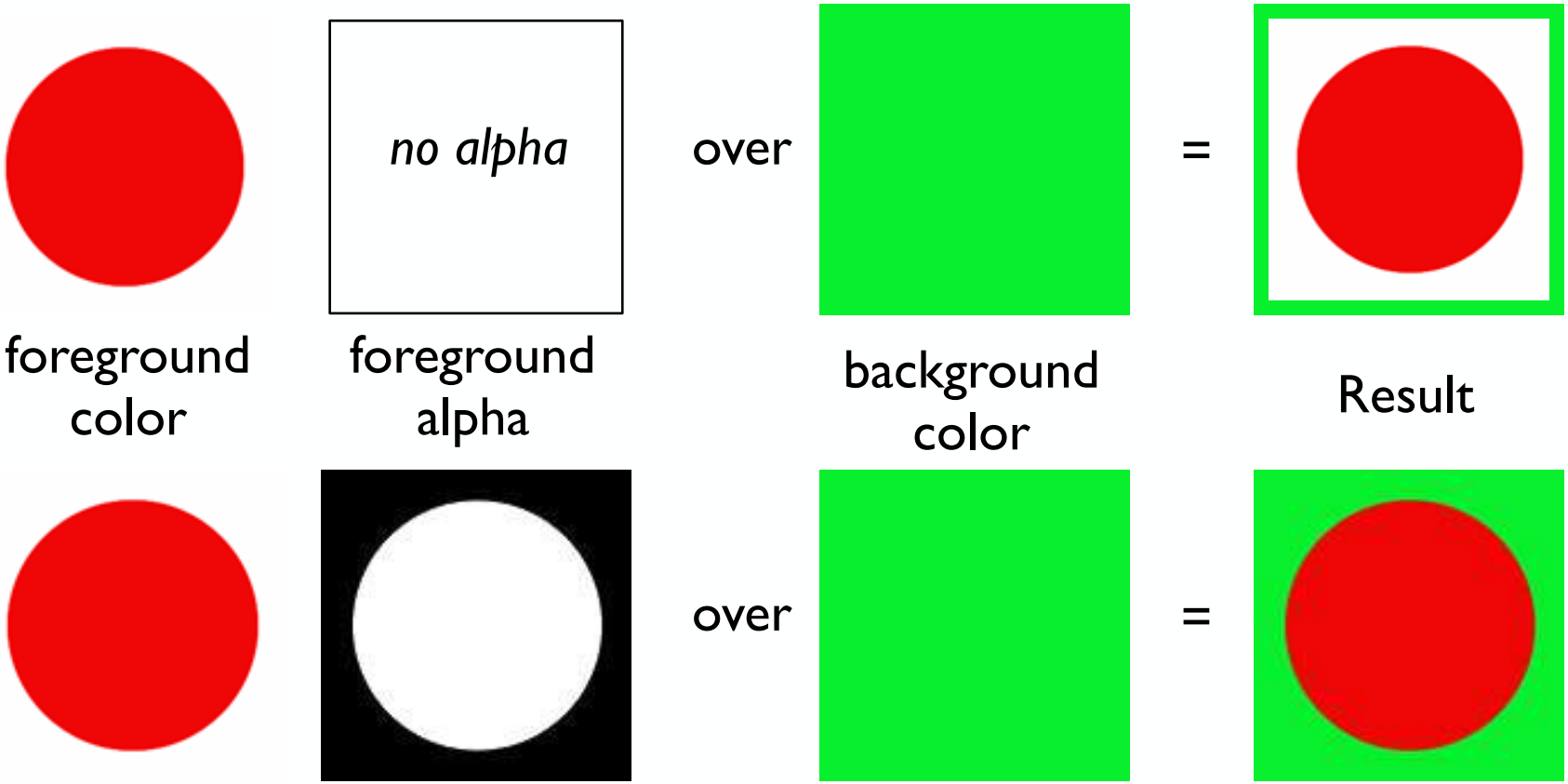
Foreground and background

- In many cases just adding is not enough
- Example: compositing in film production
  - shoot foreground and background separately
  - also include CG elements
  - this kind of thing has been done in analog for decades
  - how should we do it digitally?

© 2017 Fabio Pellacini and Steve Marschner

Compositing Images

- encode transparency for each pixel: *alpha* channel



© 2017 Fabio Pellacini and Steve Marschner

Binary image mask

- First idea: store one bit per pixel
  - answers question "is this pixel part of the foreground?"
- Switch between images based on the bit

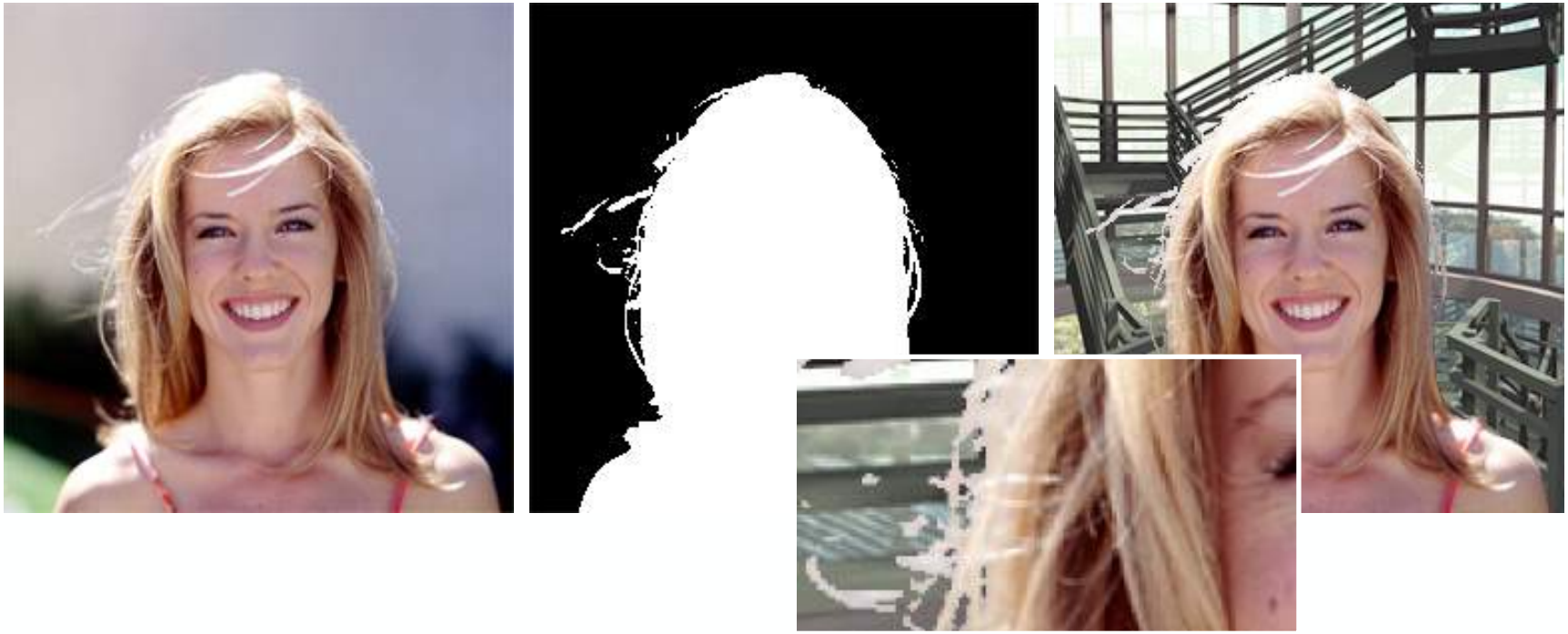
$$E = A \textbf{ over } B$$

$$\mathbf{c}_E = \begin{cases} \mathbf{c}_A & \alpha > 0 \\ \mathbf{c}_B & \alpha = 0 \end{cases}$$

© 2017 Fabio Pellacini and Steve Marschner

Binary image mask

- Causes "jaggies"

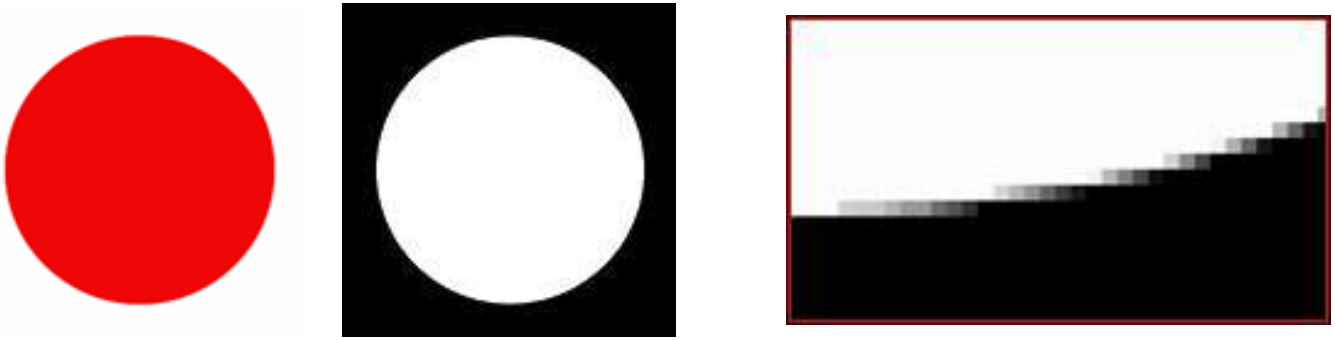


[Chuang et al. / Corel]

© 2017 Fabio Pellacini and Steve Marschner

Partial pixel coverage

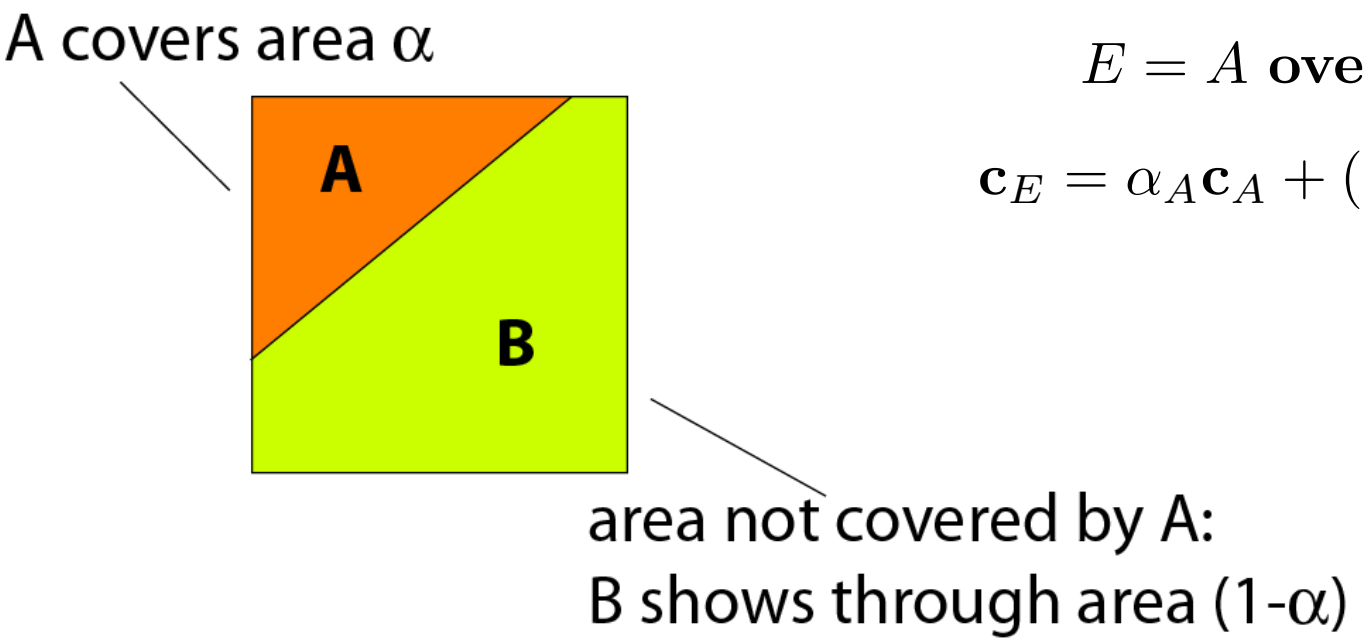
- pixels near boundary are not strictly foreground or background
  - interpolate boundary pixels between the fg. and bg. colors
  - store fractional alpha



© 2017 Fabio Pellacini and Steve Marschner

Alpha compositing

- Formalized in 1984 by Porter & Duff
- Linearly interpolate based on fractional alpha
- Efficient: 8 more bits (total 32), 2 multiplies + 1 add per pixel
- Assume A and B cover the whole pixel



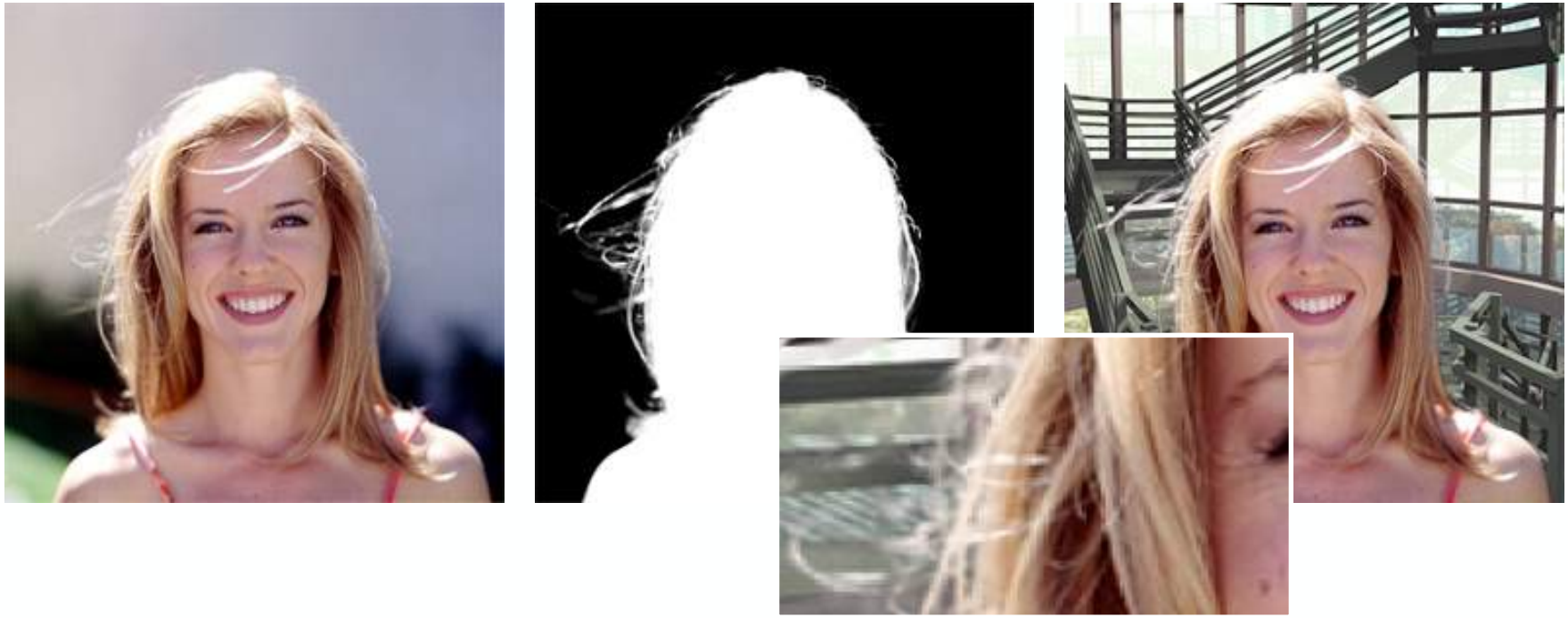
$$E = A \textbf{ over } B$$

$$\mathbf{c}_E = \alpha_A \mathbf{c}_A + (1 - \alpha_A)\mathbf{c}_B$$

© 2017 Fabio Pellacini and Steve Marschner

Alpha compositing

- Smooth transition around edges



[Chuang et al. / Corel]

© 2017 Fabio Pellacini and Steve Marschner

Compositing composites

- in real applications we have *n* layers
  - *Titanic* example
- compositing foregrounds to create new foregrounds
  - what to do with α?
- desirable property: associativity
  - can composite top-down and bottom-up

$$A \textbf{ over } (B \textbf{ over } C) = (A \textbf{ over } B) \textbf{ over } C$$

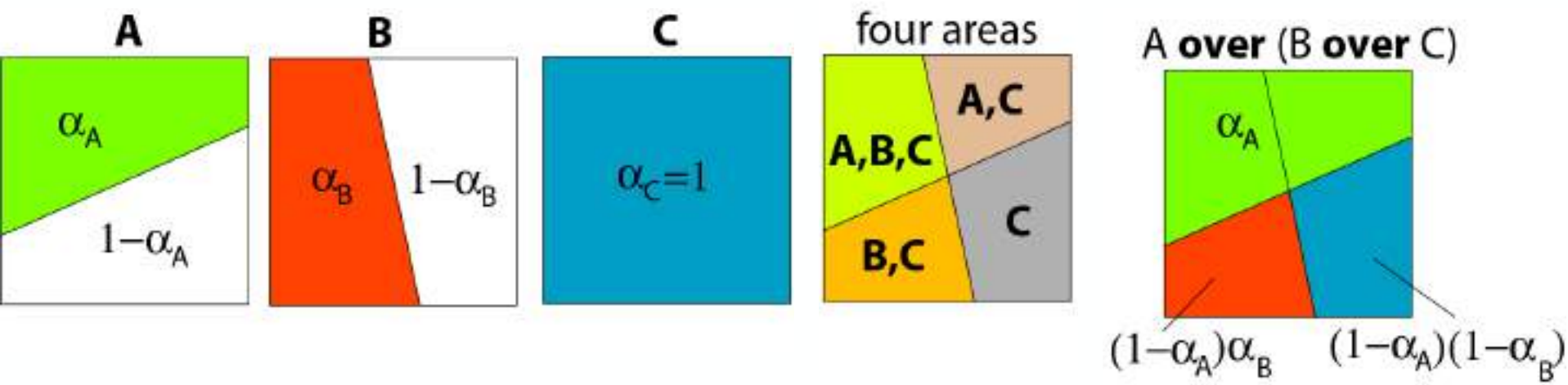
© 2017 Fabio Pellacini and Steve Marschner

Compositing composites

- Compute compositing taking into account that areas are covered by both A and B and that C covers the whole pixel

$$D = A \textbf{ over } (B \textbf{ over } C)$$

$$\begin{aligned}\mathbf{c}_D &= \alpha_A \mathbf{c}_A + (1-\alpha_A)[\alpha_B \mathbf{c}_B + (1-\alpha_B)\mathbf{c}_C] = \\ &= \alpha_A \mathbf{c}_A + (1-\alpha_A)\alpha_B \mathbf{c}_B + (1-\alpha_A)(1-\alpha_B)\mathbf{c}_C\end{aligned}$$



© 2017 Fabio Pellacini and Steve Marschner

Compositing composites

- Compute coverage of (A **over** B)

$$\alpha_{(A\ \mathbf{over}\ B)} = 1 - (1-\alpha_A)(1-\alpha_B) =$$
$$= \alpha_A + (1-\alpha_A)\alpha_B$$

- but combing colors becomes complex in D = (A **over** B) **over** C

$$\mathbf{c}_D = \alpha_A \mathbf{c}_A + (1-\alpha_A)\alpha_B \mathbf{c}_B + (1-\alpha_A)(1-\alpha_B)\mathbf{c}_C =$$
$$= \alpha_{(A\ \mathbf{over}\ B)}(\ldots) + (1-\alpha_{(A\ \mathbf{over}\ B)})\mathbf{c}_C$$

© 2017 Fabio Pellacini and Steve Marschner

Premultiplied Alpha

- Compositing equation again for $E = A \textbf{ over } B$

$$\mathbf{c}_E = \alpha_A \mathbf{c}_A + (1 - \alpha_A)\alpha_B \mathbf{c}_B$$

- Note c_A appears only in the product $\alpha_A c_A$
- Multiply it ahead of time: *premultiplied alpha*:
  - store pixel value (r', g', b', α) where $c' = \alpha c$

$$\mathbf{c}'_E = \mathbf{c}'_A + (1 - \alpha_A)\mathbf{c}'_B \text{ with } \mathbf{c}' = \alpha \mathbf{c}$$

- Generalizes to the case of C not covering the whole pixel with

$$\alpha_E = \alpha_A + (1 - \alpha_A)\alpha_B$$

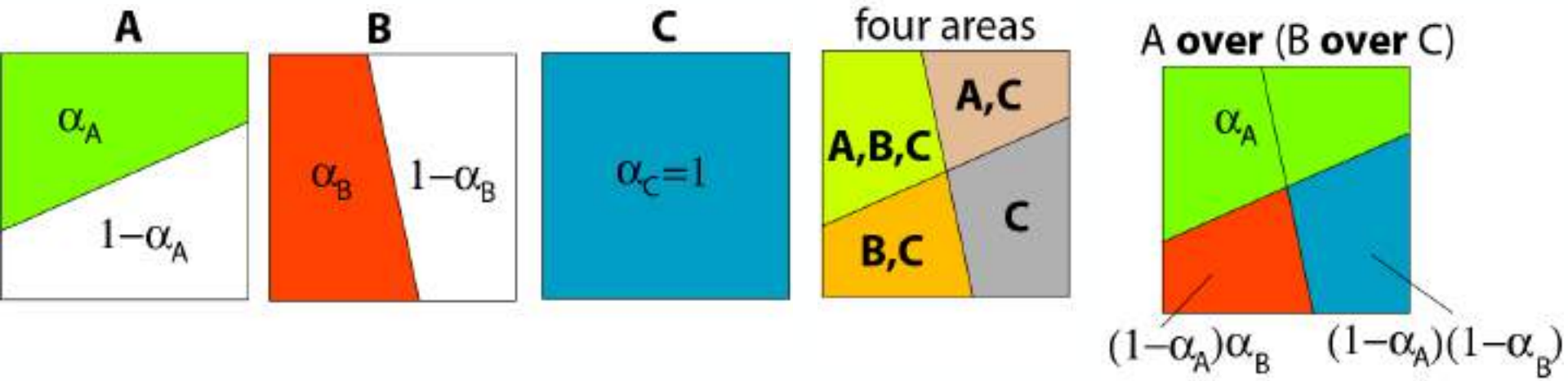
  - we will not prove this

© 2017 Fabio Pellacini and Steve Marschner

Associativity

$$
\begin{aligned}
\mathbf{c}'_D &= \mathbf{c}'_A + (1-\alpha_A)\mathbf{c}'_{(B\ \textbf{over}\ C)} = \\
&= \mathbf{c}'_A + (1-\alpha_A)[\mathbf{c}'_B + (1-\alpha_B)\mathbf{c}'_C] = \\
&= \mathbf{c}'_A + (1-\alpha_A)\mathbf{c}'_B + (1-\alpha_A)(1-\alpha_B)\mathbf{c}'_C = \\
&= [\mathbf{c}'_A + (1-\alpha_A)\mathbf{c}'_B] + (1-\alpha_A)(1-\alpha_B)\mathbf{c}'_C = \\
&= \mathbf{c}'_{(A\ \textbf{over}\ B)} + (1-\alpha_{(A\ \textbf{over}\ B)})\mathbf{c}'_C
\end{aligned}
$$

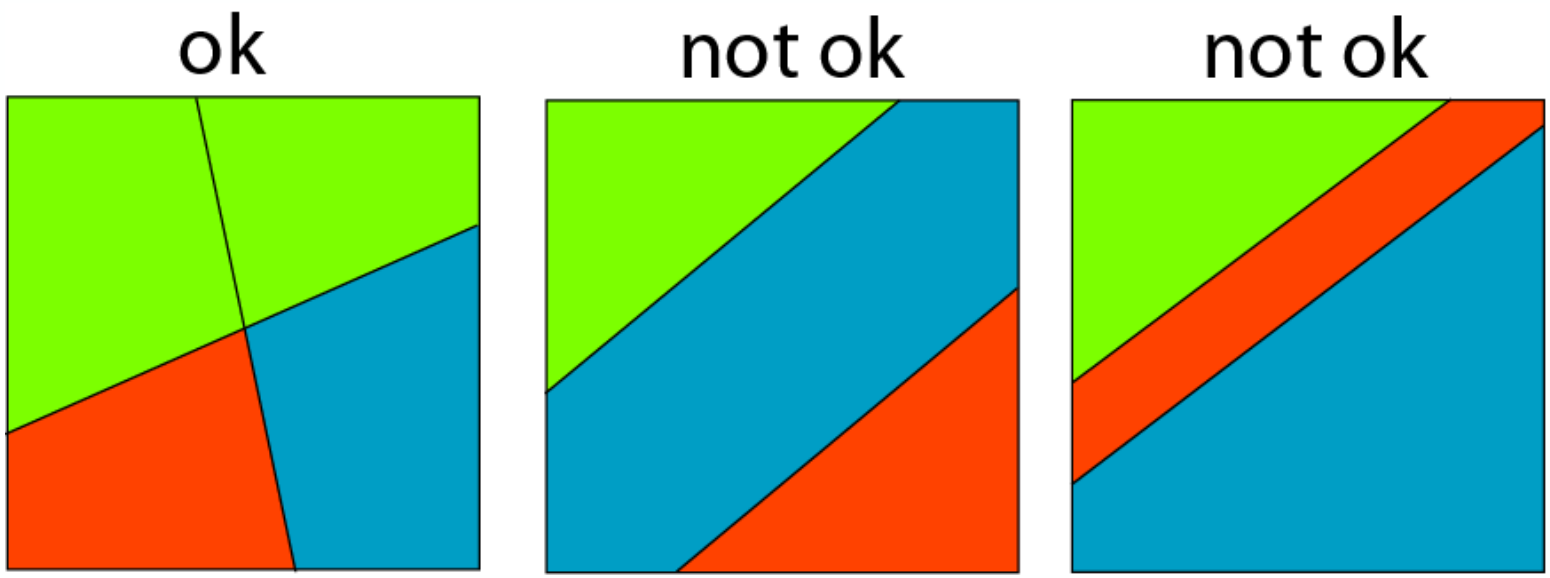
$$\Rightarrow\ (A\ \textbf{over}\ B)\ \textbf{over}\ C = A\ \textbf{over}\ (B\ \textbf{over}\ C)$$



© 2017 Fabio Pellacini and Steve Marschner

Independent coverage

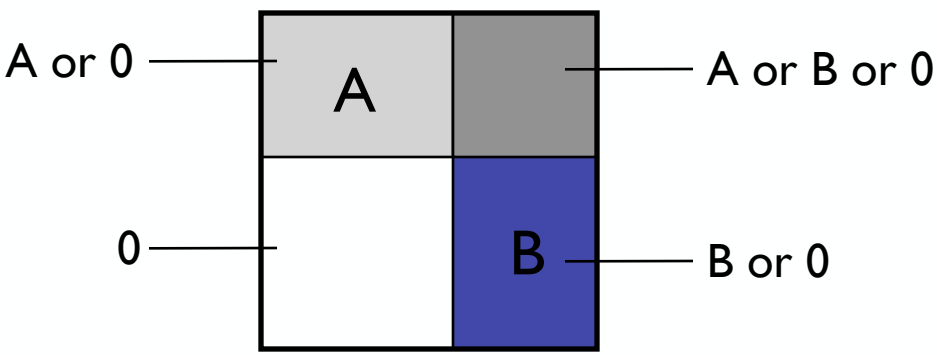
- Why is it reasonable to blend α like a color?
- Simplifying assumption: covered areas are independent
  - that is, uncorrelated in the statistical sense
- Hold in most, but not all cases



© 2017 Fabio Pellacini and Steve Marschner

Compositing Algebra

$$\alpha_E = A \textbf{ op } B$$
$$c'_E = F_A c'_A + F_B c'_B$$



1 x 2 x 3 x 2 = 12 reasonable choices

operation	quadruple	diagram	F_A	F_B
clear	(0,0,0,0)		0	0
A	(0,A,0,A)		1	0
B	(0,0,B,B)		0	1
A **over** *B*	(0,A,B,A)		1	$1-\alpha_A$
B **over** *A*	(0,A,B,B)		$1-\alpha_B$	1
A **in** *B*	(0,0,0,A)		α_B	0
B **in** *A*	(0,0,0,B)		0	α_A
A **out** *B*	(0,A,0,0)		$1-\alpha_B$	0
B **out** *A*	(0,0,B,0)		0	$1-\alpha_A$
A **atop** *B*	(0,0,B,A)		α_B	$1-\alpha_A$
B **atop** *A*	(0,A,0,B)		$1-\alpha_B$	α_A
A **xor** *B*	(0,A,B,0)		$1-\alpha_B$	$1-\alpha_A$

[Porter & Duff 84]

© 2017 Fabio Pellacini and Steve Marschner

Compositing Graphs

- Large compositing graphs are common
- Associativity allows to cache partial results
- In turn this means that we can have large graph without paying cost and at low engineering cost
  - so compositing is used everywhere
  - Photoshop
  - Web
  - PDF
  - UIs

© 2017 Fabio Pellacini and Steve Marschner

Math Review

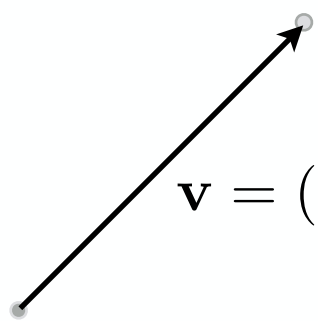
Prof. Fabio Pellacini
[original slides by Prof. Steve Marschner]

© 2020 Fabio Pellacini

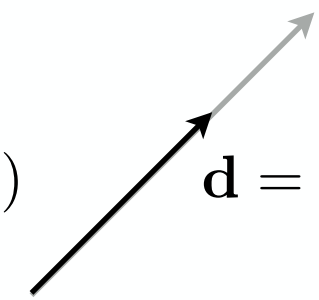
Geometric Quantities

- **Point**: location in space
- **Vector**: orientation with an associated magnitude
  - difference between points
- **Direction**: orientation with unit magnitude
- In cartesian coordinates, representation with 3 numbers

$$\mathbf{p} = (p_x, p_y, p_z)$$



$$\mathbf{v} = (v_x, v_y, v_z)$$



$$\mathbf{d} = (d_x, d_y, d_z)$$

© 2020 Fabio Pellacini

Linear Algebra Vectors

- Most graphics software uses linear algebra vectors for all points, directions and (geometric) vectors
- Indicated with 3 cartesian coordinates

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

- Typical Implementation using simple structs

```
struct vec3f {
  float x, y, z;
}
```


© 2020 Fabio Pellacini

Vector Arithmetic

- Linear algebra operations: sum and scalar product

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_x + v_x \\ u_x + v_y \\ u_z + v_z \end{bmatrix} \qquad s\mathbf{v} = \begin{bmatrix} sv_x \\ sv_y \\ sv_z \end{bmatrix}$$

- Dot product: scalar equal to the product of the vectors' lengths times the cosine between them
  - used to check whether two vectors are orthogonal

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = |\mathbf{u}||\mathbf{v}| \cos\theta$$

- Vector length: square root of the dot product

$$l = \sqrt{v_x^2 + v_y^2 + v_z^2} = \sqrt{\mathbf{v} \cdot \mathbf{v}}$$

© 2020 Fabio Pellacini

Vector Arithmetic

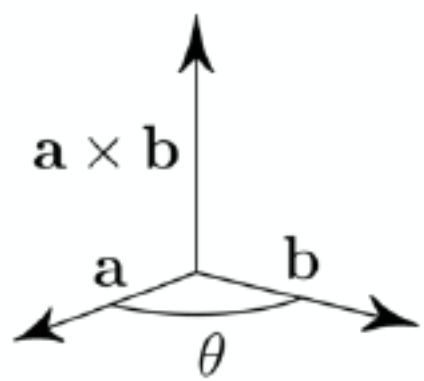
- Cross product: vector orthogonal to the operands, whose length is equal to the product of the vectors' lengths times the sine between them
- Used to construct orthonormal vectors

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

$$|\mathbf{u} \times \mathbf{v}| = |\mathbf{u}||\mathbf{v}| \sin\theta$$

$$(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{u} = 0$$

$$(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{v} = 0$$



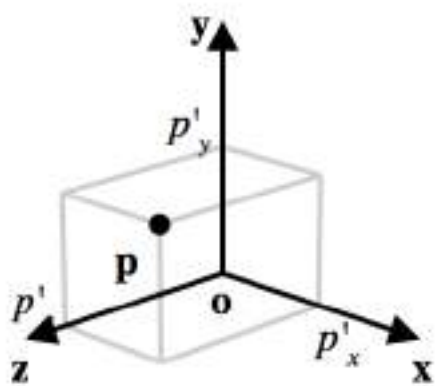
© 2020 Fabio Pellacini •

Coordinate Systems

- Points, vectors, directions are defined w.r.t. a coordinate system
- Coordinate systems, or frames, specify an origin and three axis
  - if not defined, we intend the "unit" or "world" frame

$$F = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}\}$$

```
struct frame3f {
  vec3f x, y, z, o;
}
```

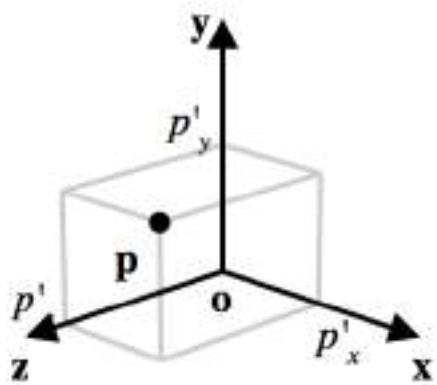


© 2020 Fabio Pellacini

Coordinate Systems

- Given a frame defined w.r.t world space, we can transform a point coordinates to/from the coordinate frame
- World to local: subtract origin, project onto the frame axes
- Local to world: add origin to the scaled frame axes

$$\mathbf{p}^l = \begin{bmatrix} (\mathbf{p}-\mathbf{o})\cdot\mathbf{x} \\ (\mathbf{p}-\mathbf{o})\cdot\mathbf{y} \\ (\mathbf{p}-\mathbf{o})\cdot\mathbf{z} \end{bmatrix} \qquad \mathbf{p} = \mathbf{o} + p_x^l\mathbf{x} + p_y^l\mathbf{y} + p_z^l\mathbf{z}$$

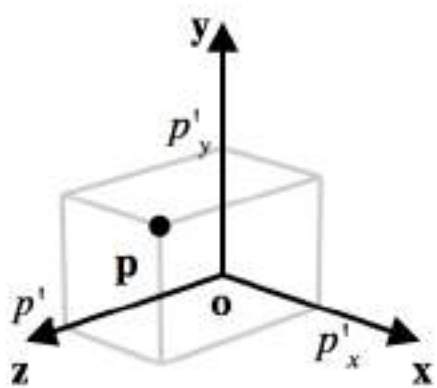


© 2020 Fabio Pellacini

Coordinate Systems

- Vector transforms as difference of points
- Boils down to ignoring the origin

$$\mathbf{v}^l = \begin{bmatrix} \mathbf{v} \cdot \mathbf{x} \\ \mathbf{v} \cdot \mathbf{y} \\ \mathbf{v} \cdot \mathbf{z} \end{bmatrix} \qquad \mathbf{v} = v_x^l \mathbf{x} + v_y^l \mathbf{y} + v_z^l \mathbf{z}$$



© 2020 Fabio Pellacini

Matrices

- Consider only 3x3 matrices for now

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

- Write in column notation

$$M = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix} = \begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix}$$

© 2020 Fabio Pellacini

Matrices

- Matrix-vector multiply: weighted sum of matrix' columns

$$M\mathbf{v} = v_x\mathbf{x} + v_y\mathbf{y} + v_z\mathbf{z}$$

- Matrix-matrix multiply: columns are matrix-vector multiplies

$$MM' = \left[M\mathbf{x}', M\mathbf{y}', M\mathbf{z}'\right]$$

- Matrix inverse
  - analytic for small matrices, numerical for all others

$$MM^{-1} = M^{-1}M = I$$

© 2020 Fabio Pellacini

Matrices

- Transpose: flip the row and columns

$$M = \left[m_{ij}\right] \rightarrow M^T = \left[M_{ji}\right]$$

- Orthonormal matrices: columns are orthogonal to each other and normalized

$$M_o = [\mathbf{c}_i] \Leftrightarrow \mathbf{c}_i \cdot \mathbf{c}_j = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases}$$

- Orthonormal inverse is its transpose

$$M_o^{-1} = M_o^T$$

© 2020 Fabio Pellacini

Matrices and Frames

- Define a matrix whose columns are the coordinate frame axes

$$F = \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}\} \quad M = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix}$$

- Matrix is orthonormal
- Express coordinate transformations with algebra operations

$$\mathbf{p} = M\mathbf{p}^l + \mathbf{o} \qquad \mathbf{v} = M\mathbf{v}^l$$

$$\mathbf{p}^l = M^T(\mathbf{p} - \mathbf{o}) \qquad \mathbf{v}^l = M^T\mathbf{v}$$

© 2020 Fabio Pellacini

Constructing Frames

- Make a vector orthogonal to another by subtracting its projection

$$\mathbf{u}_\perp = \mathbf{u} - (\mathbf{u} \cdot \mathbf{v})\mathbf{v}$$

- Create a frame from **z**' and **y**'
  - normalize **z'**
  - make **y'** orthogonal to **z** and normalize it
  - compute **x** as cross product of **z** and **y**

$$\mathbf{z} = \frac{\mathbf{z}'}{|\mathbf{z}'|} \quad \mathbf{y} = \frac{\mathbf{y}' - (\mathbf{y}' \cdot \mathbf{z})\mathbf{z}}{|\mathbf{y}' - (\mathbf{y}' \cdot \mathbf{z})\mathbf{z}|} \quad \mathbf{x} = \mathbf{z} \times \mathbf{y}$$

© 2020 Fabio Pellacini

Implicit curves

- Equation to tell whether we are on the curve or surface

$$\{\mathbf{v} \mid f(\mathbf{v}) = 0\}$$

- Example: line (orthogonal to u, distance k from 0)

$$\{\mathbf{v} \mid \mathbf{v} \cdot \mathbf{u} + k = 0\}$$ (u is a unit vector)

- Example: circle (center p, radius r)

$$\{\mathbf{v} \mid (\mathbf{v} - \mathbf{p}) \cdot (\mathbf{v} - \mathbf{p}) - r^2 = 0\}$$

- Always define boundary of region
  - (if f is continuous)

© 2020 Fabio Pellacini

Explicit curves

- Also called parametric
- Equation to map domain into plane

$$\{f(t) \mid t \in D\}$$

- Example: line (containing p, parallel to u)

$$\{\mathbf{p} + t\mathbf{u} \mid t \in \mathbb{R}\}$$

- Example: circle (center b, radius r)

$$\{\mathbf{p} + r[\cos t \;\; \sin t]^T \mid t \in [0, 2\pi)\}$$

- Like tracing out the path of a particle over time
- Variable *t* is the "parameter"

© 2020 Fabio Pellacini

Algebra-vs-geometry views

- In the previous slides we often changed between algebraic notation to geometric concepts
- This is very common in graphics
- In general, we tend to use algebraic concepts as much as possible since they are simpler to handle and easier to implement

© 2020 Fabio Pellacini