Introduction to the theory of 3D Computer Graphics

Geometric modeling of 3D objects

wireframe modeling
surface modeling: assembling surface primitives, parametric/implicit surfaces
volume modeling: assembling volume primitives, matrix of voxels

In depth: polynomial parametric curves and surfaces

Principle
Bezier curves and surfaces
B-spline curves and surfaces, NURBS

Light modeling

light representation
physical phenomena: reflection and refraction
estimating light intensities with Phong model
normal vector of a polygonal surface - Lambert, Gouraud, Phong methods

Rendering

Z buffer algorithm
ray tracing algorithm
ray marching algorithm for an implicit surface

Advanced rendering

global illumination problem
backward ray tracing
radiosity

Textures

2D textures, mapping, aliasing, anti-aliasing
3D textures

Procedural modeling

fractal landscape

Animation

principle - degrees of freedom of a scene
kinematic animation
dynamic animation
animation of articulated structures

References:

Advanced Animation and Rendering Techniques – Theory and Practice
Alan Watt, Mark Watt; Addison-Wesley, ACM press

http://www.mactech.com/articles/develop/issue_25/schneider.html (for NURBS)

Numerical Recipes in C: The Art of Scientific Computing
William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery;
Cambridge University Press (for Runge Kutta method)

Geometric modeling of 3D objects

how to represent an object ?

by a grid of lines → *wireframe modeling*

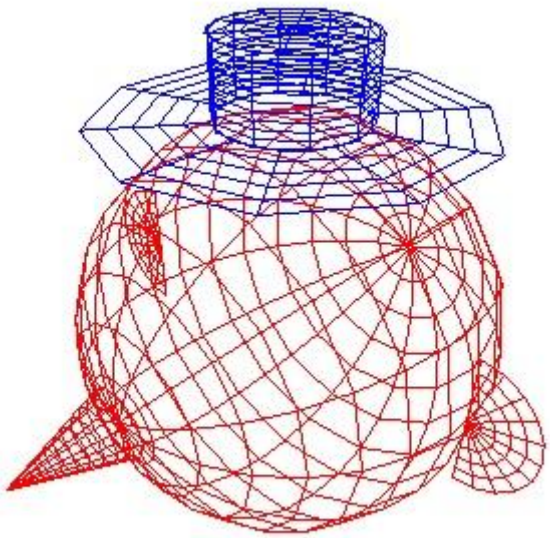
by the surface which delimits it → *surface modeling*

by the volume it occupies → *volume modeling*

Wireframe modeling

set of joined facets → shared *vertices*

lines representing the facets' borders



only the lines are drawn → should we draw the lines corresponding to the hidden parts of the surface ?

Surface modeling

- *surface primitive:* = basic surface:
  - polygon (triangle, quadrangle)
  - sphere, spherical cap
  - disc, conical surface, cylindrical surface
  - polynomial parametric surface (Bezier, spline)

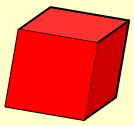
assembling surface primitives:

- the actual surface of an object is an assembly of several surface primitives

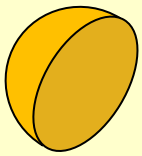
→ continuity constraints:

- the resulting surface must be entirely closed
  = *C_0 continuity* at the edge between two joining primitives
- if smooth surface requested, continuity of the surface normal
  = *C_1 continuity* at the edge between two joining primitives
- eventually continuity of the curvature
  = *C_2 continuity* at the edge between two joining primitives

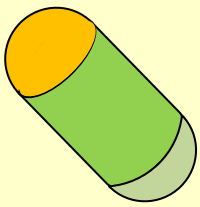
examples:



cube = assembly of 6 quadrangles → C_0



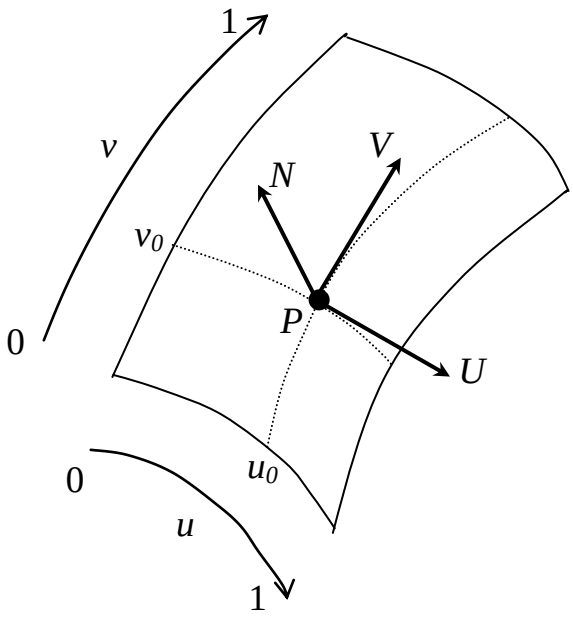
half-sphere = spherical cap closed with a disk → C_0



capsule: cylindrical surface closed with two spherical caps → C_1

parametric representation of a surface primitive:

by varying parameters *u* and *v*, we cover the whole extent of the surface primitive:



- coordinates of *point P* (*u* , *v*) of surface: $$P(u,v)=\begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix} \;;\; u \in I_u \;, v \in I_v$$

- *tangent vectors* to the surface at point $P(u_0, v_0)$: $$\begin{cases} U(u_0,v_0)=\dfrac{\partial P(u_0,v_0)}{\partial u} \\ V(u_0,v_0)=\dfrac{\partial P(u_0,v_0)}{\partial v} \end{cases}$$

- *normal vector* at point $P(u_0, v_0)$: $$N(u_0,v_0)=U(u_0,v_0)\times V(u_0,v_0)$$

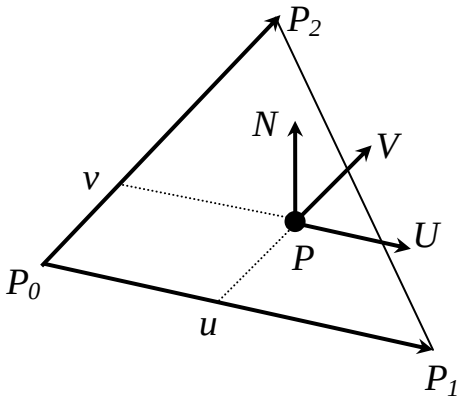
$$\begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix}=\begin{bmatrix} U_x \\ U_y \\ U_z \end{bmatrix}\times\begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix}=\begin{bmatrix} U_y\cdot V_z-U_z\cdot V_y \\ U_z\cdot V_x-U_x\cdot V_z \\ U_x\cdot V_y-U_y\cdot V_x \end{bmatrix}$$

N is used to compute the light intensity at point *P*

N is normalized at 1 → N becomes a **unit vector**: $$N \leftarrow \frac{N}{\|N\|}$$

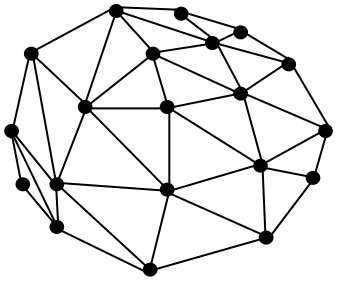
- parametric representation of a *triangle*:

$$\begin{cases} P(u,v) = P_0 + u \cdot (P_1 - P_0) + v \cdot (P_2 - P_0) \\ u, v \in [0,1] \\ u + v \le 1 \end{cases}$$

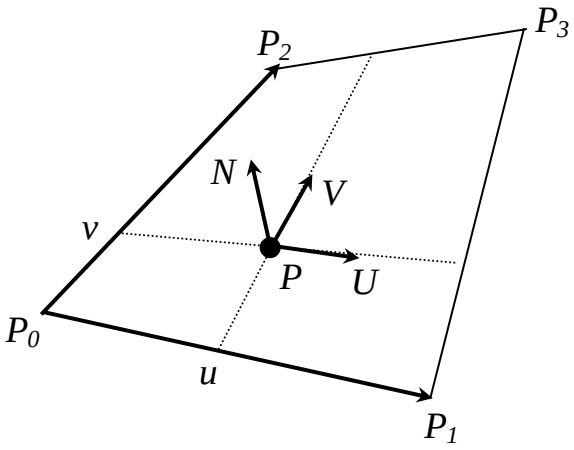


- *data structures* for a surface defined by a set of triangles:

  - array of vertices: vertex[i] = { x , y , z }
  - array of triangles: triangle[j] = { iv_0 , iv_1 , iv_2 }



- parametric representation of a *quadrangle*:

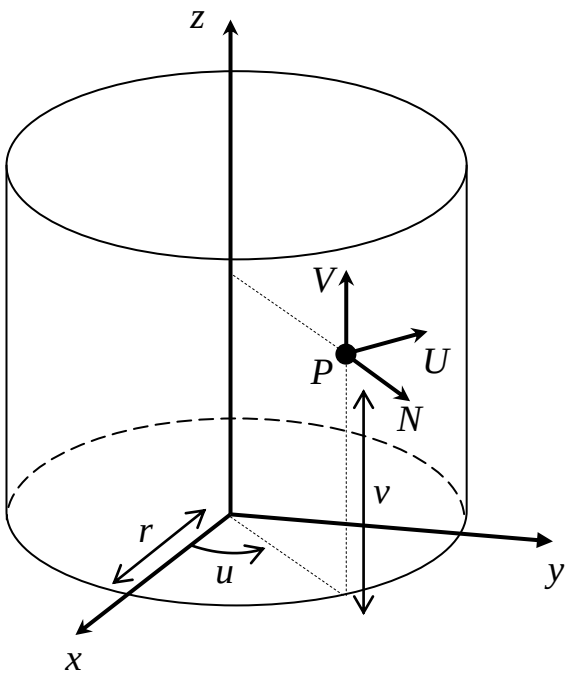


$$\begin{cases} P(u,v) = (1-u) \cdot (1-v) \cdot P_0 + u \cdot (1-v) \cdot P_1 + v \cdot (1-u) \cdot P_2 + u \cdot v \cdot P_3 \\ u, v \in [0,1] \end{cases}$$

this surface may be twisted!

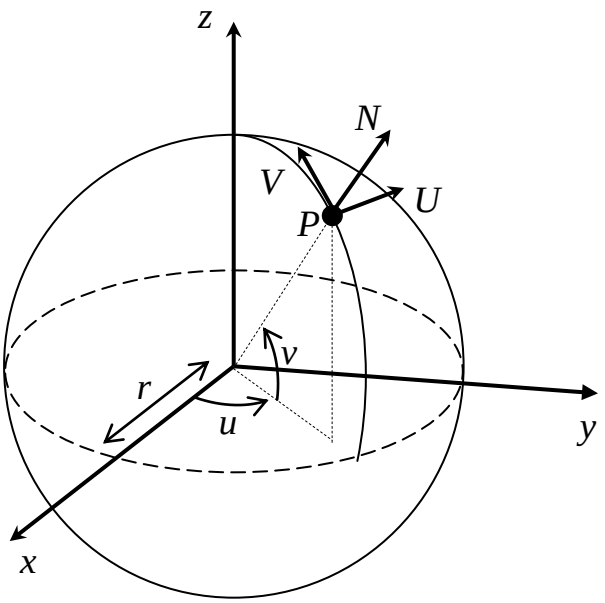
- parametric representation of a *cylinder*:

$$\begin{cases} P(u,v) = \begin{bmatrix} r\cdot\cos u \\ r\cdot\sin u \\ v \end{bmatrix} \\ u\in[0,2\pi] \quad v\in[h_{\min},h_{\max}] \end{cases}$$



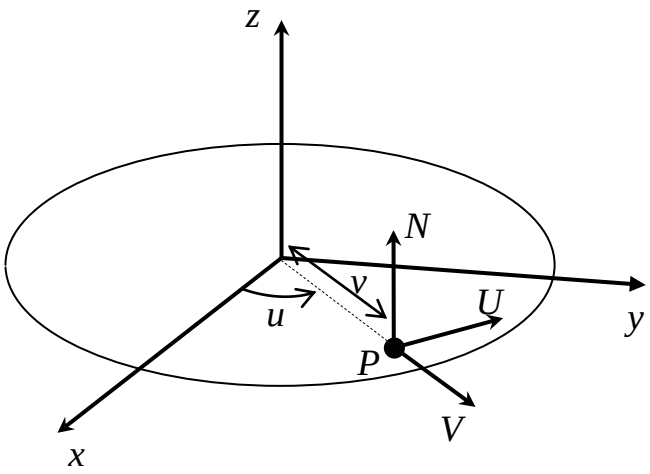
- parametric representation of a *sphere*:

$$\begin{cases} P(u,v) = \begin{bmatrix} r\cdot\cos v\cdot\cos u \\ r\cdot\cos v\cdot\sin u \\ r\cdot\sin v \end{bmatrix} \\ u\in[0,2\pi] \quad v\in\left[-\frac{\pi}{2},\frac{\pi}{2}\right] \end{cases}$$



- parametric representation of a *disk*:

$$\begin{cases} P(u,v) = \begin{bmatrix} v\cdot\cos u \\ v\cdot\sin u \\ 0 \end{bmatrix} \\ u\in[0,2\pi] \quad v\in[0,r] \end{cases}$$

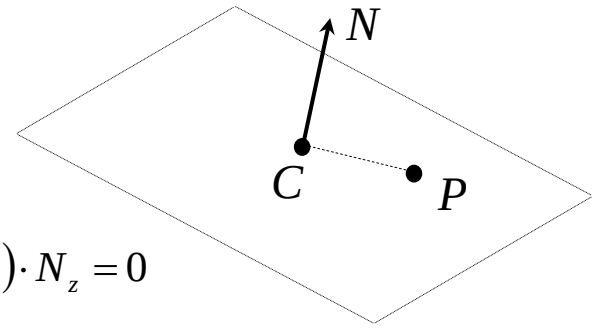


implicit representation of a surface primitive:

a more compact mathematical definition than parametric representation:

P(*x*,*y*,*z*) ∈ surface ⇔ implicit equation over position *x*,*y*,*z*: $f(x, y, z) = 0$

- implicit representation of an *unlimited plane:*



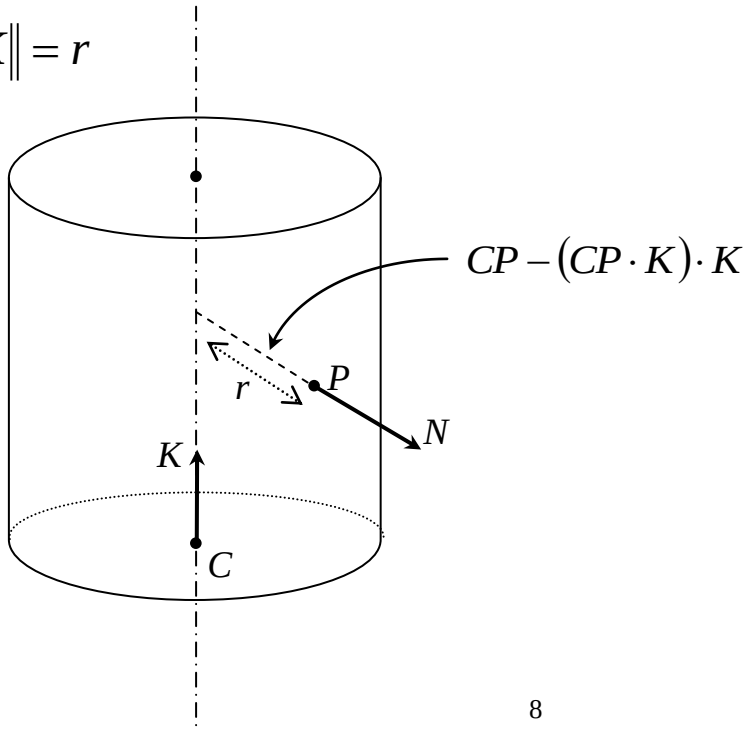
$$CP \cdot N = 0 \Leftrightarrow (x - x_C) \cdot N_x + (y - y_C) \cdot N_y + (z - z_C) \cdot N_z = 0$$

- implicit representation of a *sphere*:

$$dist(C, P) = r \Leftrightarrow (x - x_C)^2 + (y - y_C)^2 + (z - z_C)^2 = r^2$$

- implicit representation of an *unlimited cylinder*: (axis: point C, unit vector K)

$$dist(axis, P) = r \Leftrightarrow \left\| CP - (CP \cdot K)K \right\| = r$$

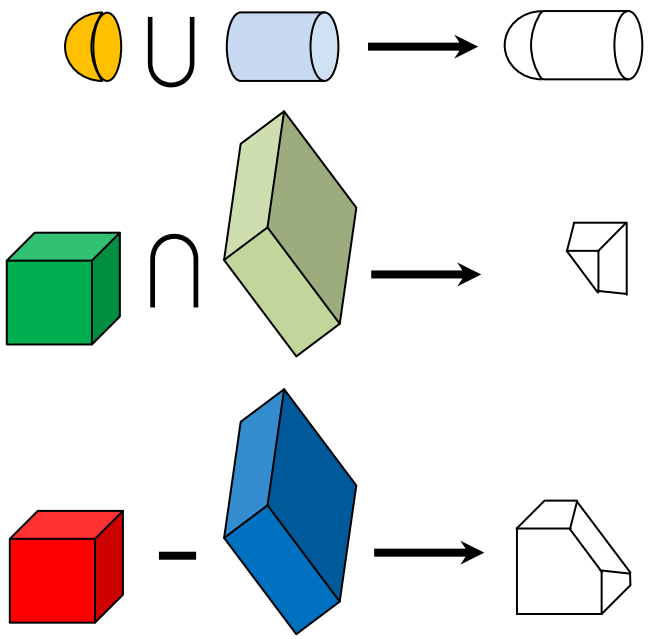


Volume modeling

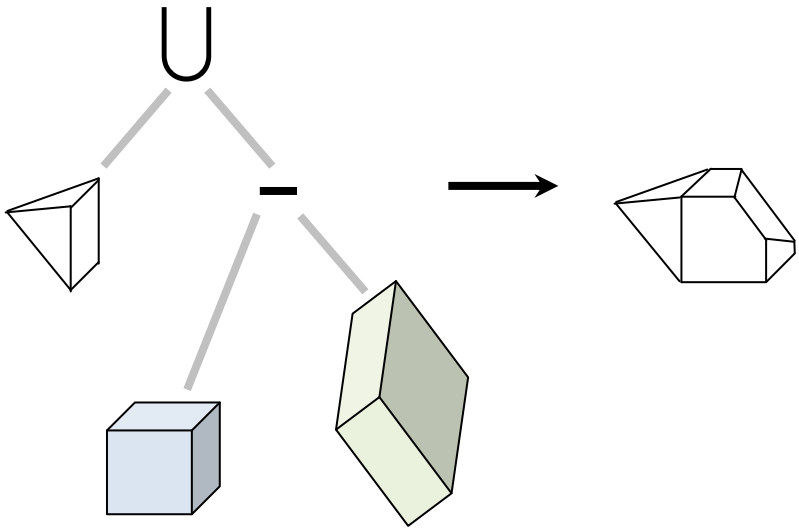
assembling volume primitives - Constructive Solid Geometry:

- *volume primitive*: sphere, cylinder, conic, cube...

- *union*, *intersection* and *cut* operators:



- hierarchical combination of the operators:

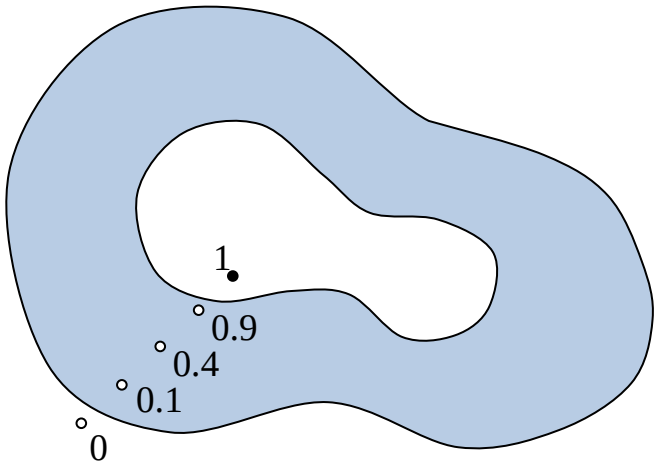


volume modeling with 3D density field:

generalization of implicit equation $f(x, y, z) = 0$ for implicit surface:

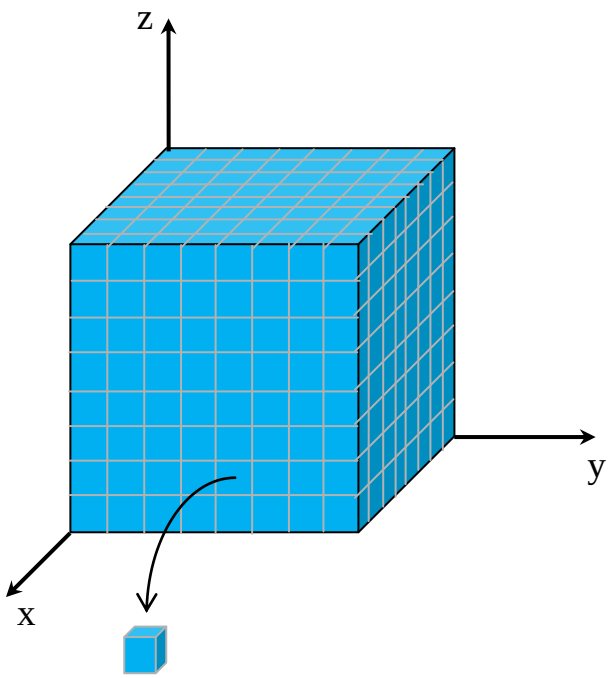
$$0 < f(x,y,z) \leq 1$$

→ object with a partially transparent border:



volume modeling with matrix of voxels:

- *voxel* = volume element (*pixel* = picture element)



drawbacks:

- huge memory required
- totally impractical for the operator to initialize the voxels himself

→ in general, the voxels are directly filled by raw data obtained from a sensing device or resulting from a numerical simulation (fluid dynamics)

example: Medical Imagery

In depth: polynomial parametric curves and surfaces

Principle

- a set of *control points* is used to define the curve / surface

  the curve or surface passes through a control point or is simply "attracted" by it

- coordinates of a point of the curve / surface = *weighted sum* of the control points' coordinates

  the weighting coefficients are polynomial functions of *parameter u* (for a curve)
  or of *parameters u, v* (for a surface)

curve: $n + 1$ control points

$$\begin{cases} P(u) = \sum_{i=0}^{i=n} P_i \cdot F_i(u) \\ u \in [0,1] \end{cases}$$

surface: = *patch* $(n + 1) \times (n + 1)$ control points

$$\begin{cases} P(u,v) = \sum_{i=0}^{i=n} \sum_{j=0}^{j=n} P_{ij} \cdot F_i(u) \cdot F_j(v) \\ u \in [0,1] \quad v \in [0,1] \end{cases}$$

- if $F_i(u)$, $F_j(v)$ polynomials are of degree m, the patches compounding a complex surface can be adjusted so that continuity between them is C_{m-1}

  in general, $m = 3$ → continuity C_2 = continuity of surface, normal and curvature

Bezier curves and surfaces

Bezier curves:

- n+1 control points
  the functions are of degree $m = n$

→ a Bezier curve undergoes the global influence of all its control points

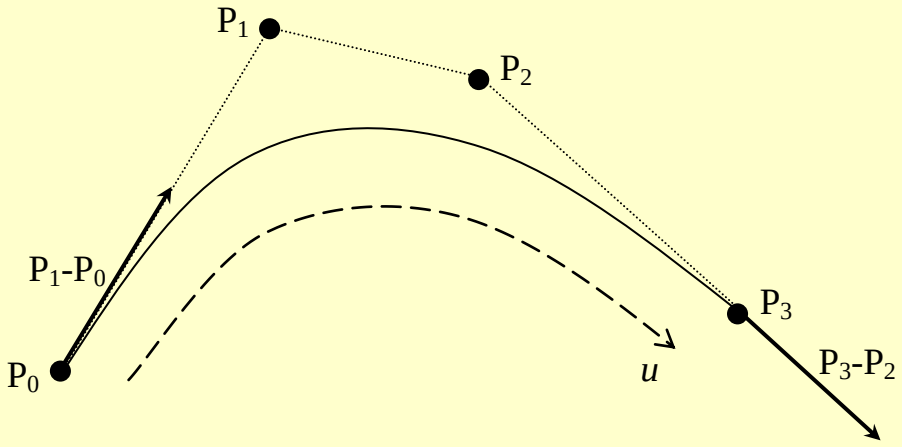
→ the number of control points is very limited because
polynomials whose degree is too high are too cumbersome

$$\begin{cases} P(u) = \sum_{i=0}^{i=n} P_i \cdot B_i^n(u) \\ u \in [0,1] \\ B_i^n(u) = C_i^n \cdot u^i \cdot (1-u)^{n-i} \end{cases}$$

with $C_i^n = \dfrac{n!}{i! \cdot (n-i)!}$

example: $m = n = 3$ → 4 control points

$$\begin{cases} B_0^3(u) = (1-u)^3 \\ B_1^3(u) = 3u \cdot (1-u)^2 \\ B_2^3(u) = 3u^2 \cdot (1-u) \\ B_3^3(u) = u^3 \end{cases}$$



the curve passes through P_0 and P_3

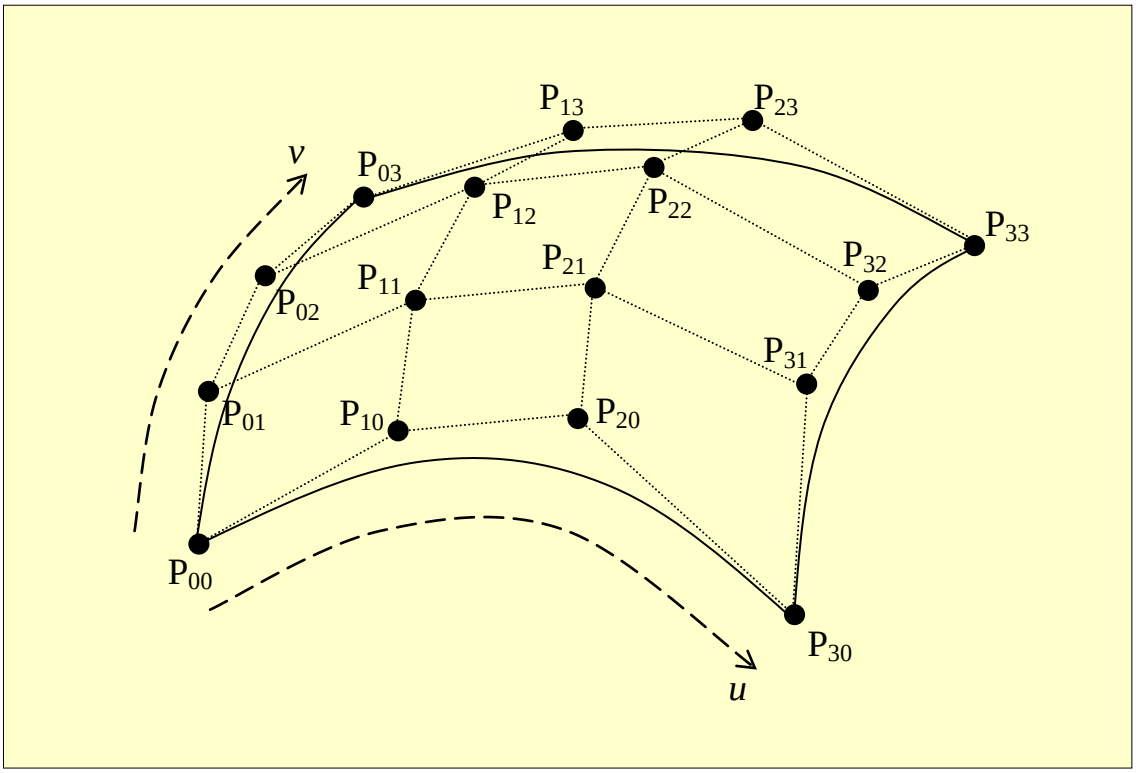
P_1 and P_2 help adjust the curve by warping it

P_1 and P_2 help define the curve tangents at P_0 and P_3

Bezier surfaces:

$$\begin{cases} P(u,v) = \sum_{i=0}^{i=n} \sum_{j=0}^{j=n} P_{ij} \cdot B_i^n(u) \cdot B_j^n(v) \\ u \in [0,1] \quad v \in [0,1] \end{cases}$$

example: $m = n = 3$ → 4×4 control points



it is difficult in practice to join together several Bezier patches because of the continuity constraints

B-spline curves and surfaces, NURBS

B-spline curves: *Basis spline*

- n+1 control points
  the functions are of degree m << n

  → we can easily afford a huge number of control points

  → a given point of the curve undergoes only the local influence of the m+1 closest control points

$$\begin{cases} P(u) = \sum_{i=0}^{i=n} P_i \cdot N_i^m(u) \\ u \in [0,1] \end{cases}$$

- we want $C_{m-1} = C_2$ continuity constraint all along the curve
  even though the polynomials are of limited degree

- the B-spline curve is subdivided into n curve segments:

  $0 = u_0 < u_1 < \ldots < u_{n-1} < u_n = 1$

  if values u_i equally spaced → *uniform* B-spline
  else *non-uniform* B-spline

- different weights associated to the control points → *rational* B-spline
  else *non-rational* B-spline

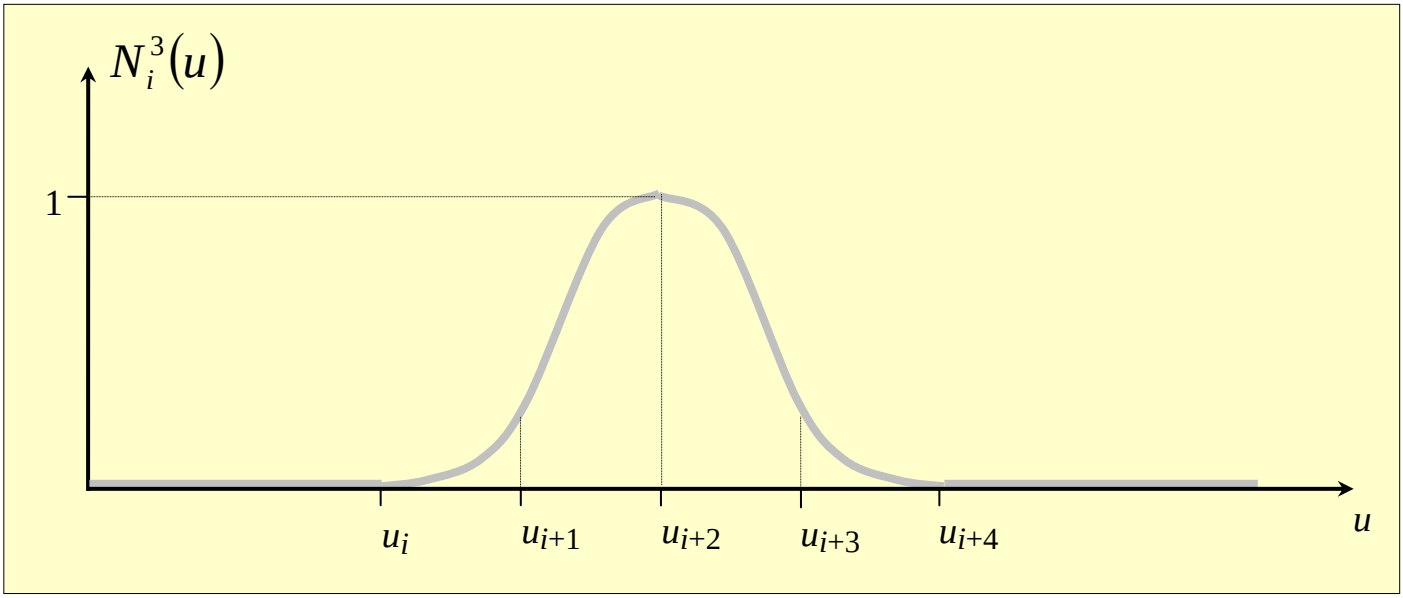
- *NURBS* = *Non-Uniform Rational Basis-Spline*

- the $N_i^m(u)$ polynomials are defined recursively:

$$\begin{cases} N_i^0(u) = 1 \text{ if } u \in [u_i, u_{i+1}] \text{ , 0 else} \\ N_i^m(u) = \dfrac{u - u_i}{u_{i+m} - u_i} N_i^{m-1}(u) + \dfrac{u_{i+m+1} - u}{u_{i+m+1} - u_{i+1}} N_{i+1}^{m-1}(u) \end{cases}$$

$N_i^m(u)$ is non null only within $m + 1$ segments

example: cubic uniform B-spline ($m = 3$)



B-spline surfaces:

$$\begin{cases} P(u,v) = \sum_{i=0}^{i=n} \sum_{j=0}^{j=n} P_{ij} \cdot N_i^m(u) \cdot N_j^m(v) \\ u \in [0,1] \quad v \in [0,1] \end{cases}$$

- a point of the surface is influenced only by the $(m + 1) \times (m + 1)$ closest control points

Light modeling

Light representation

- *monochromatic* = only one wavelength = one sinusoid function

- in general, sum of many sinusoid functions with different wavelengths

- in practice, only 3 wavelengths considered (the eye has only 3 types of photoreceptor cells)

  → colors *Red*, *Green*, *Blue* = 3 *fundamental colors*

- computer screens usually display only the 3 fundamental colors

  → 3 parameters are needed to describe light

RGB space:

light intensities for Red , Green , Blue colors

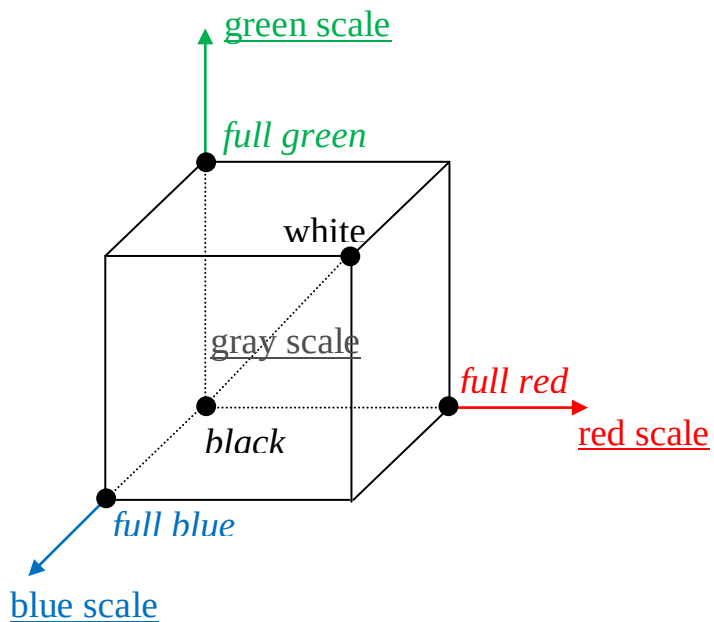
= between 0 (no intensity) and 1 (maximal intensity that the screen can display)

light intensity: $I = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$

$red = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ $green = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ $blue = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

$black = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ $white = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

$gray = \begin{bmatrix} x \\ x \\ x \end{bmatrix}$ with $0 < x < 1$ (*gray scale*)

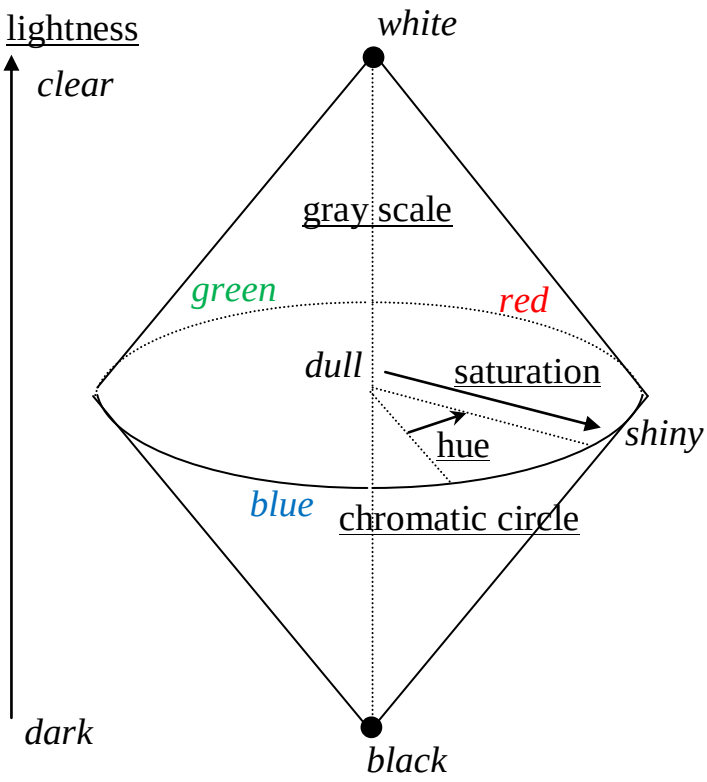


HLS space:

Hue: color on the chromatic circle

Lightness: dark to clear

Saturation: gray to full color = fraction (0 to 1) of pure color relative to gray



HLS space is a very convenient tool for the manual modeling of optical properties of surfaces

RGB space is used in computations

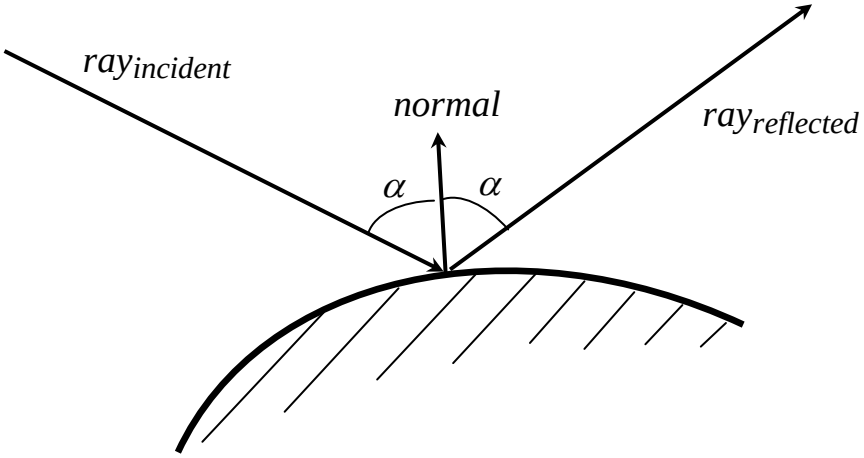
Physical phenomena: reflection and refraction

light travels through space undisturbed (if no fog ...)

but, when light touches an object, it can be either *reflected* or *refracted*

specular reflection:

one incident ray → only one reflected ray: reflection angle = incidence angle

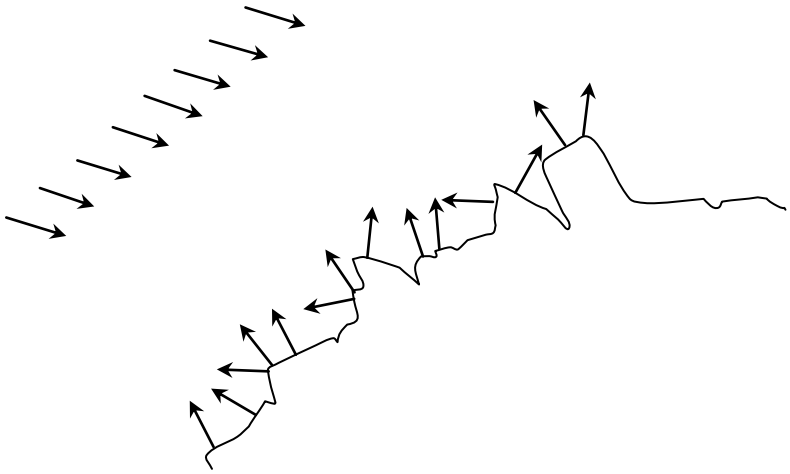


we have a perfect specular reflection only with a mirror

diffuse reflection:

in practice, surfaces are not as flat and smooth as a mirror

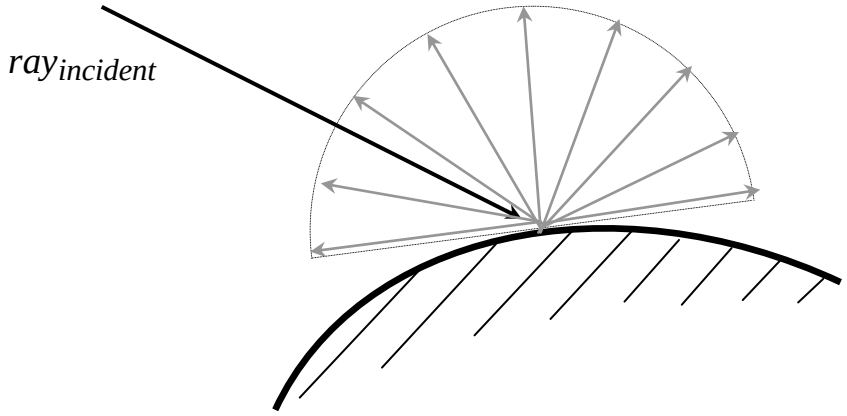
→many microscopic defaults (surface roughness) which scatter the reflected ray



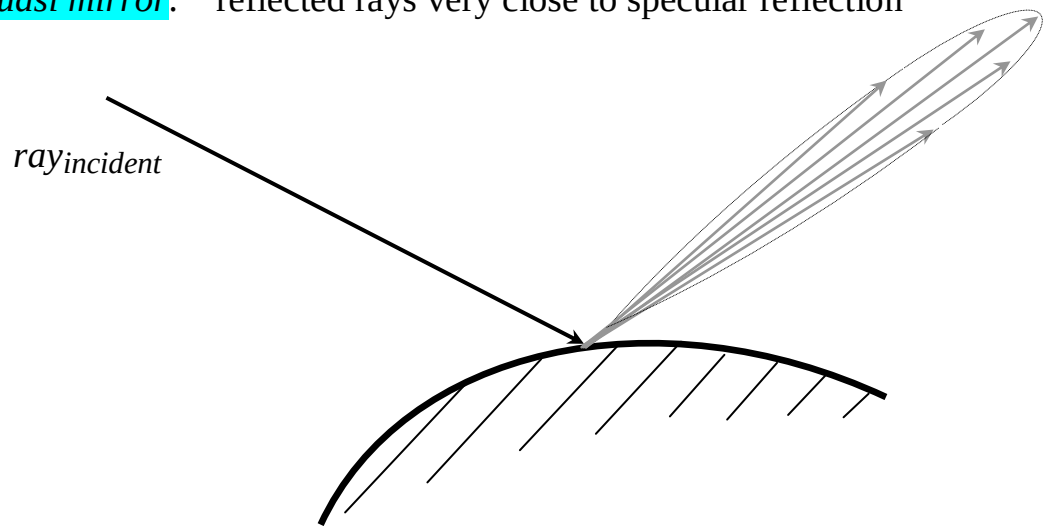
one incident ray → infinity of reflected rays, in all directions of the semi-space

diverse types of reflecting surfaces:

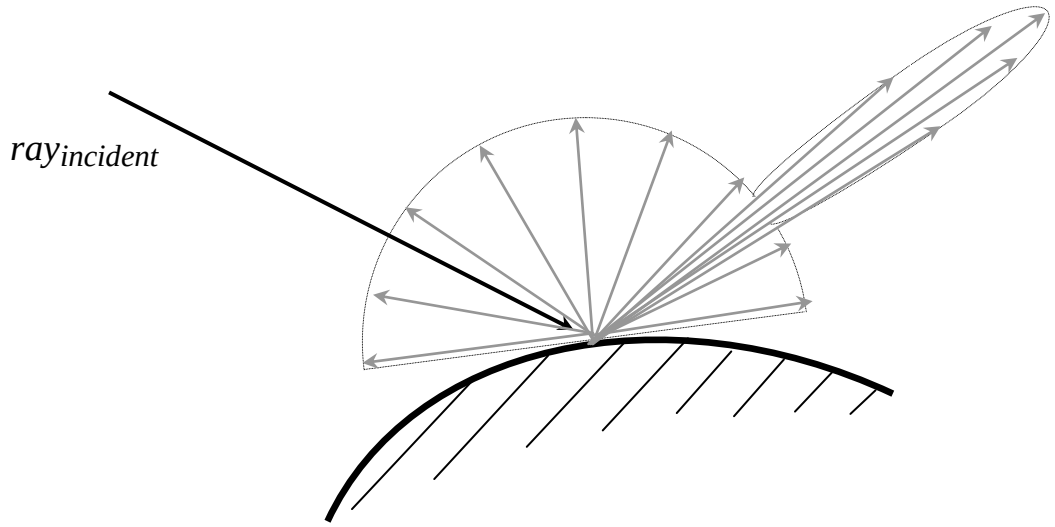
- *unpolished surface*: wood, concrete, cloth, … →reflected light is constant in all directions



- *quasi mirror*: reflected rays very close to specular reflection



- *metallic surface*: mixture of diffuse and quasi specular reflection



refraction:

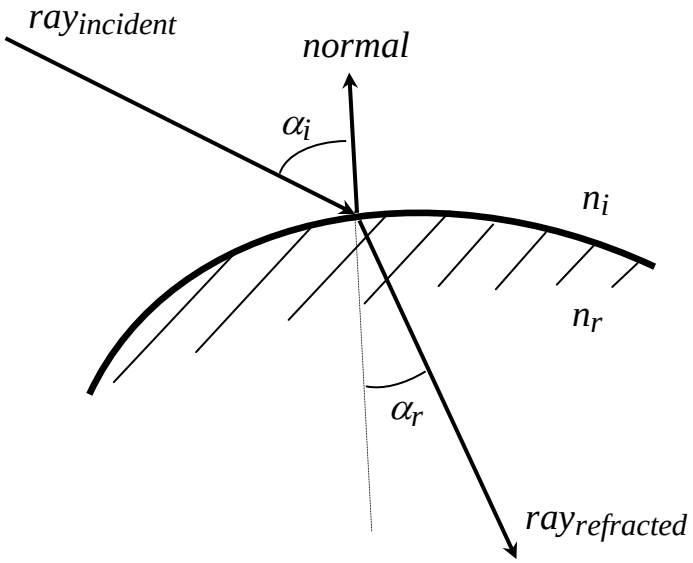
an incident ray enters a more or less transparent object and is transformed into a refracted ray

in practise, only specular refraction: one incident ray → only one refracted ray

refraction angle follows *law of refraction*:

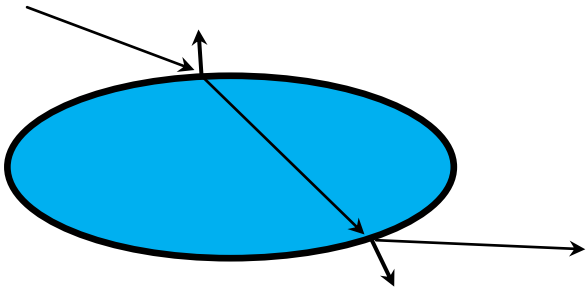
$$\frac{\sin(\alpha_r)}{\sin(\alpha_i)} = \frac{n_i}{n_r}$$

with n_i , n_r refraction indexes
(n=1 for air, n>1 inside object)



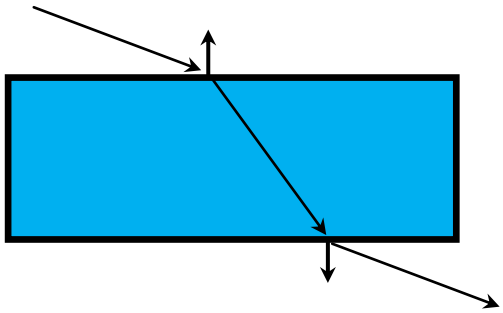
- *if refraction index constant inside the object:*

what matters are the entry and exit points of the ray (between them the ray does not vary)



- *if the two frontiers of the object are parallel:*

the direction of light propagation remains globally unchanged (but there is a shift):



Estimating light intensities with Phong model

Phong model:

total light = ambiant light
+ direct diffuse reflection light
+ direct specular reflection light

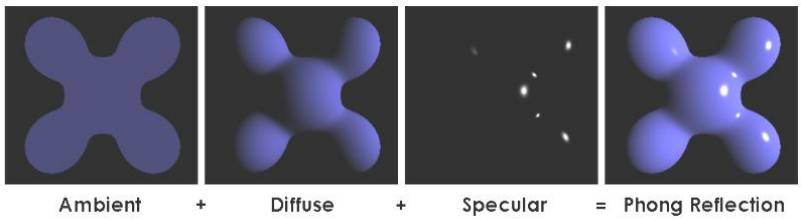


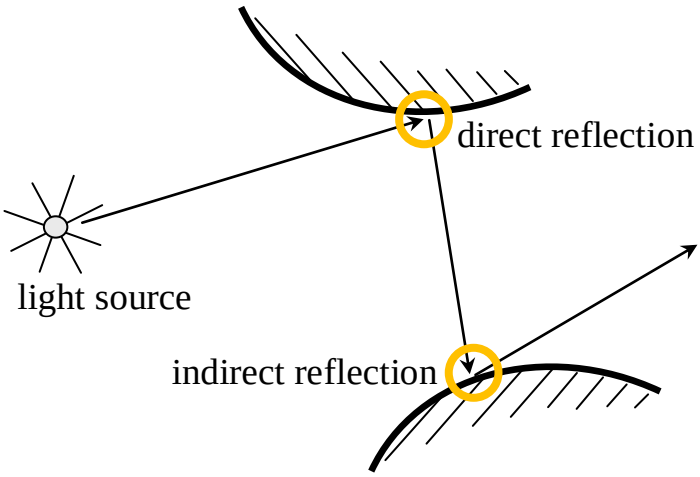
Figure courtesy of Brad Smith, Wikipedia

important simplifications to allow fast light calculations:

- only direct reflections → ambiant light is constant all over the scene
- light source is punctual (eventually several punctual light sources)
- no refraction (will be modeled with ray tracing)

diffuse reflection:

direct diffuse reflection: the incident ray comes directly from the light source



Note: all indirect reflections globally yield the *ambient light*
= many interactions between surfaces (difficult to compute, requires radiosity)
→ ambient light is assumed constant over the whole scene…

specular reflection:

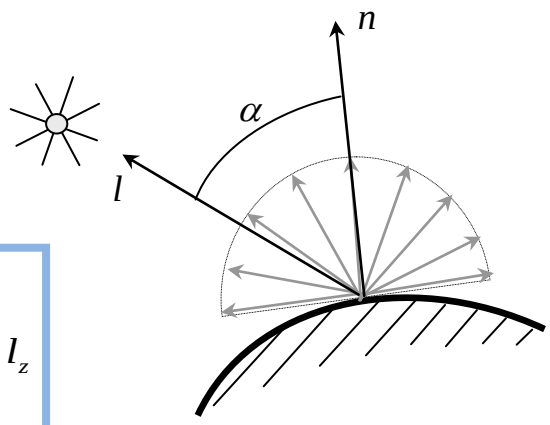
direct specular reflection: the incident ray comes directly from the light source

= simple reflection of the image of the light source over the surface

we must render the image of a light source that is not exactly punctual (else source not visible)

computing direct diffuse reflection light:

l direction to light source
n , l unit vectors



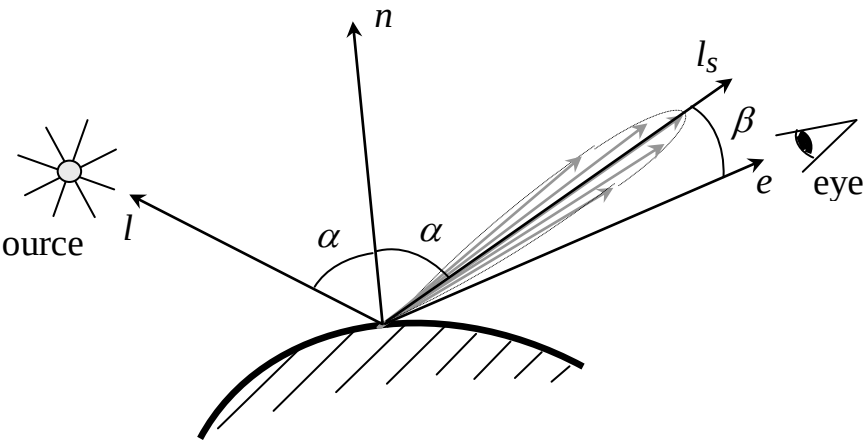
$$I_d = K_d^{refl} \cos(\alpha) = K_d^{refl} (n \cdot l)$$

$$I_d = \begin{bmatrix} {}^R K_d^{refl} \cdot q_d \\ {}^G K_d^{refl} \cdot q_d \\ {}^B K_d^{refl} \cdot q_d \end{bmatrix} \quad \text{with} \quad q_d = n_x \cdot l_x + n_y \cdot l_y + n_z \cdot l_z$$

Note:
- maximal when lighting normal to the surface
- null when lighting tangent to the surface

computing direct specular reflection:

l direction to light source
e direction to virtual eye
l_s dir. of specular reflection of light source
l_s , e unit vectors



if light source were really punctual :
- if $e = l_s$: total specular reflection
- if $e \neq l_s$ (even very slightly): no specular reflection at all → nothing visible

in practice, the light source to has a small size:

the more e is close to l_s → the higher the specular reflection

→ empirical formula: $I_s = K_s^{refl} \cos^p(\beta) = K_s^{refl} (e \cdot l_s)^p$ with p very big...

→ $$I_s = \begin{bmatrix} {}^R K_s^{refl} \cdot q_s \\ {}^G K_s^{refl} \cdot q_s \\ {}^B K_s^{refl} \cdot q_s \end{bmatrix} \quad \text{with} \quad q_s = \left(e_x \cdot l_s^x + e_y \cdot l_s^y + e_z \cdot l_s^z\right)^p$$

ambient light is constant:

$$I_a = K^{amb} = \begin{bmatrix} {}^{R}K^{amb} \\ {}^{G}K^{amb} \\ {}^{B}K^{amb} \end{bmatrix}$$

computing total light:

$$I = K^{amb} + K_d^{refl}(n \cdot l) + K_s^{refl}(e \cdot l_s)^p$$

→

$$I = \begin{bmatrix} {}^{R}K^{amb} + {}^{R}K_d^{refl} \cdot q_d + {}^{R}K_s^{refl} \cdot q_s \\ {}^{G}K^{amb} + {}^{G}K_d^{refl} \cdot q_d + {}^{G}K_s^{refl} \cdot q_s \\ {}^{B}K^{amb} + {}^{B}K_d^{refl} \cdot q_d + {}^{B}K_s^{refl} \cdot q_s \end{bmatrix}$$

with $\begin{cases} q_d = n_x \cdot l_x + n_y \cdot l_y + n_z \cdot l_z \\ q_s = (e_x \cdot l_s^x + e_y \cdot l_s^y + e_z \cdot l_s^z)^p \end{cases}$

if several light sources (index *i*): $I = K^{amb} + K_d^{refl} \sum_i (n \cdot l_i) + K_s^{refl} \sum_i (e \cdot l_s^i)^p$

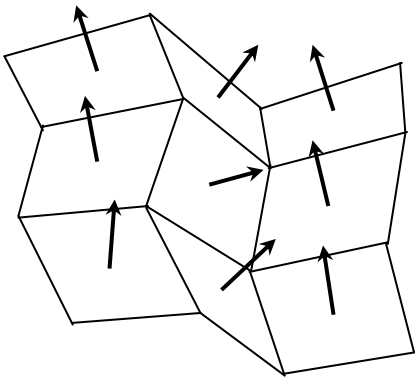
Normal vector of a polygonal surface - Lambert, Gouraud, Phong methods

- surface made up of joined polygons

→ how to give the illusion of a smooth surface?

Lambert method:

the normal is assumed constant over each polygon

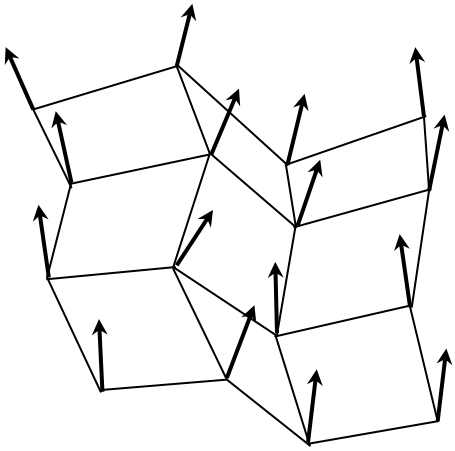


very fast but crude: the polygons are clearly visible

Gouraud method:

the normal is given at each vertex between the polygons

→ light intensity computed at the vertices
and interpolated over each point of the polygons

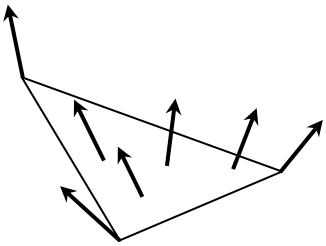


decent quality but costly because of the interpolations

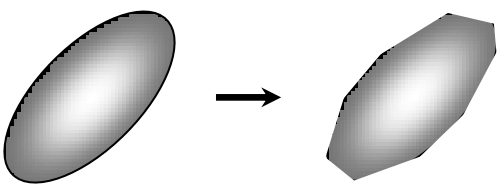
Phong method:

the normal is given at each vertex between the polygons

→ 3 coordinates of normal directly interpolated
over each point of the polygons,
then light intensity computed at each point



excellent quality but very costly...
and the borders of the object still remain crude:



Rendering

Z buffer algorithm

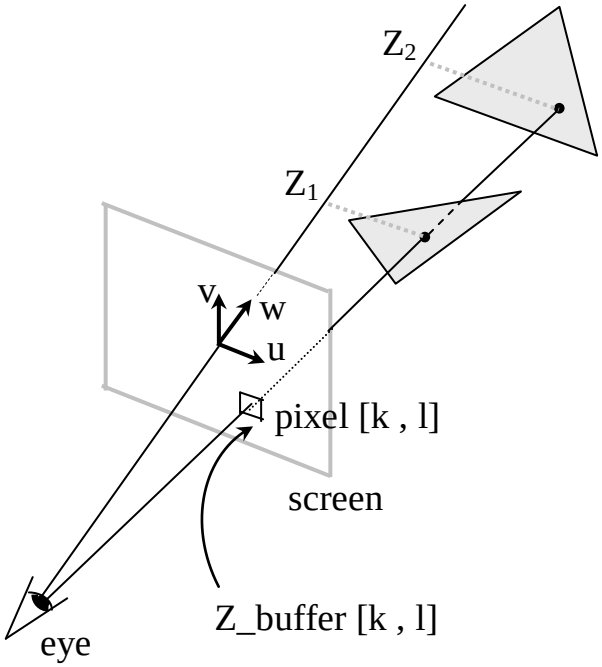
- surface made up of joined polygons

- each polygon is projected on the screen

  → the pixels inside the projected polygon are assigned the light intensity of the polygon (if Lambert method...)

- *hidden surface removal*: only the (parts of) polygons closest to the screen are displayed
  the (parts of) polygons hidden behind them are discarded

to each pixel is assigned a buffer = Z value for the currently displayed projection



if the next projected polygon contains pixel [k , l]
and new Z value < Z_buffer [k , l]

→ replace the previous projection on the pixel with the new projection

Ray tracing algorithm

- Phong model already takes into account the direct specular and direct diffuse reflections

  → ray tracing also takes into account the indirect specular reflections and refractions

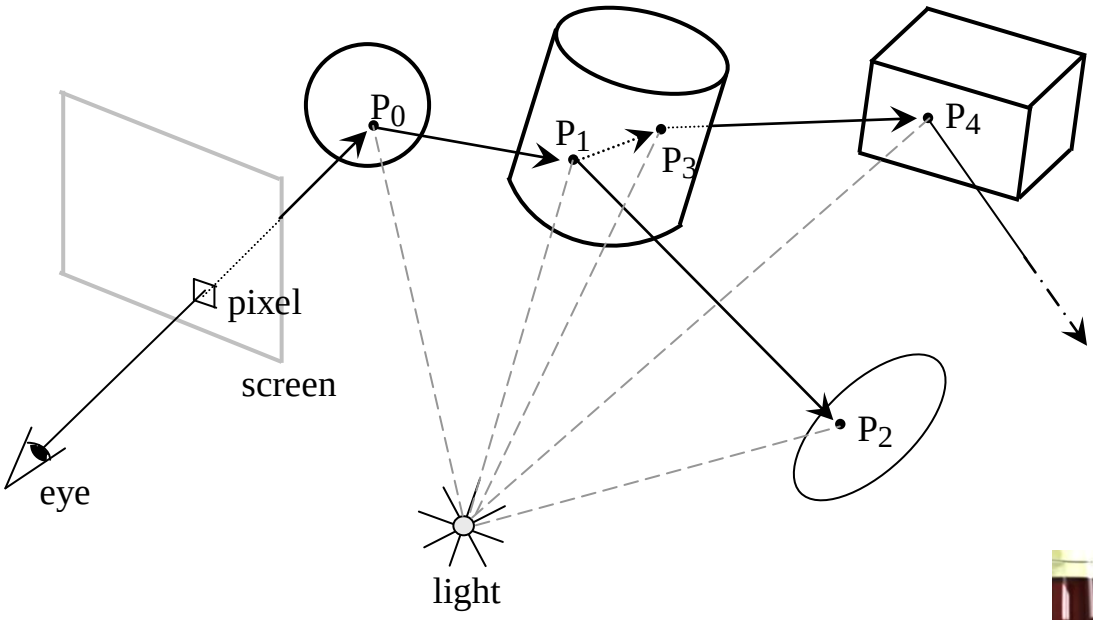
indirect diffuse reflections are still approximated as constant ambiant light
(use radiosity for accurate computation)

- in principle, we should compute all the light rays starting from the light source and being specularly reflected through the scene

  = only very few of them actually reach the virtual eye !

  → in practice, we select these relevant rays by following their invert paths

  = for each pixel, we trace a ray starting from the eye, passing through the pixel and being specularly reflected and refracted within the scene:



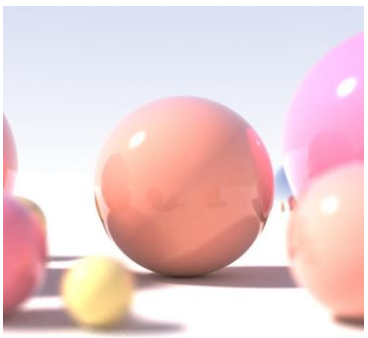


Figure courtesy of Tim Babb, Wikipedia

- first, built invert geometric paths of selected light rays
- then, compute their light intensities by following the normal direction of light propagation

Figure courtesy of Gilles Tran, Wikipedia

- light of ray starting from P_i and moving toward the eye is function of
  - direct (diffuse or specular) reflections at P_i = "local lighting" = initial Phong model
  - light intensities of rays reaching P_i through indirect specular reflections and refractions

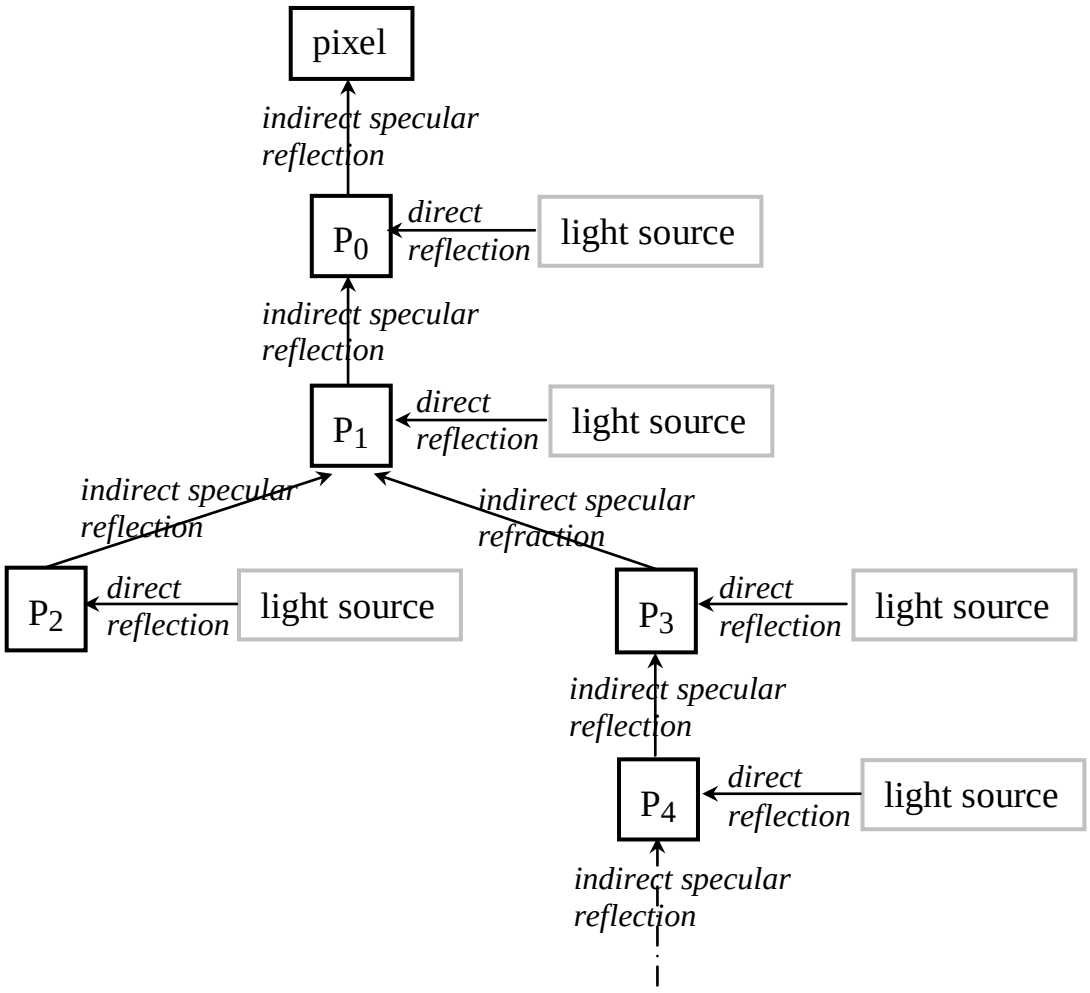
→ formula:

$$I = K^{amb} + K_d^{refl}(n \cdot l) + K_s^{refl}(e \cdot l_s)^p + K_s^{refl} I_{refl\_ray} + K_s^{refr} I_{refr\_ray}$$

$0 \le K^{refl}{}_s \le 1$ and $0 \le K^{refr}{}_s \le 1$

$K^{refl}{}_s = 0$ → no specular reflection ; $K^{refl}{}_s = 1$ → complete specular reflection
$K^{refr}{}_s = 0$ → no specular refraction ; $K^{refr}{}_s = 1$ → complete specular refraction

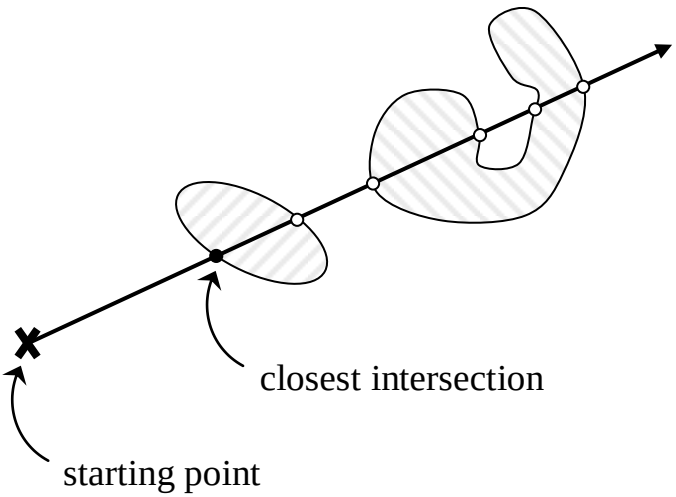
→ apply formula recursively over all specular rays that contribute to the lighting of the pixel:



basic task of ray tracing: compute the intersection of a geometric ray with all objects
= very costly if many objects...

- *hidden surface removal with ray tracing:*

  select the closest intersection to the starting point of the ray:

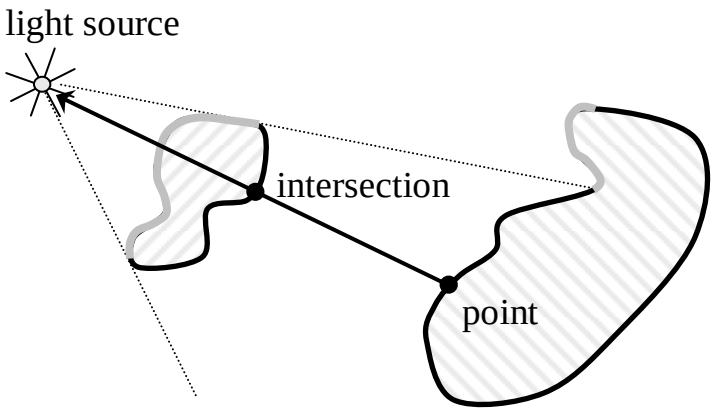


- *implementing shadows with ray tracing:*

  does a point of a surface directly receive light from the light source?

  = draw a ray between this point and the light source

  → if this ray intersects any object, the point is shadowed
  else the point is not shadowed

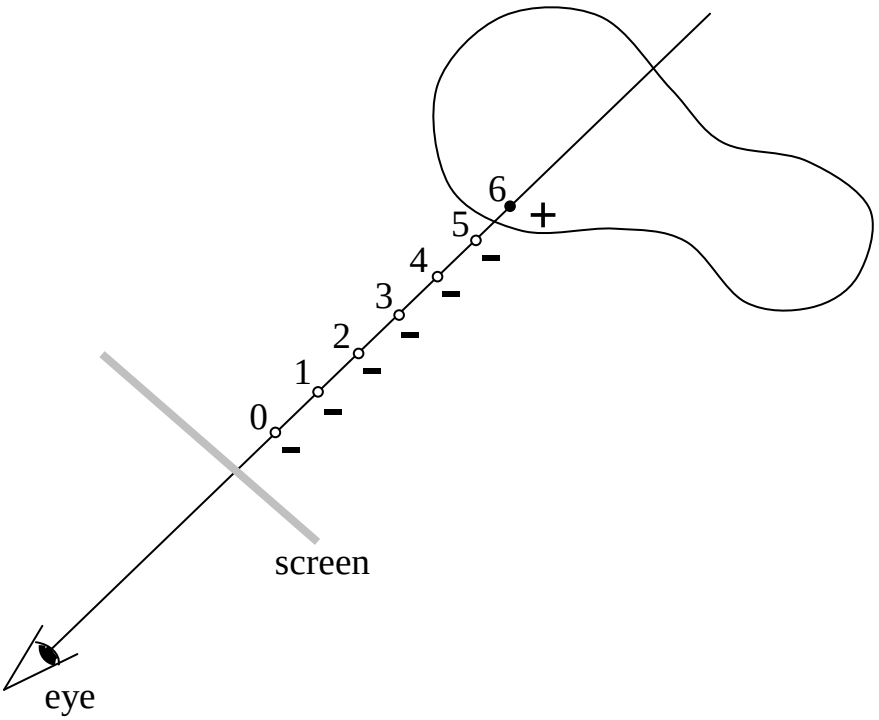


Ray marching algorithm for an implicit surface:

- *Objective:* find the intersection of a ray with a complex implicit surface $f(x,y,z)=0$

  outside the object: $f(x,y,z)<0$
  inside the object: $f(x,y,z)>0$

- *principle:*
  - "walk" step by step on the ray
  - at each step compute the value of $f(x,y,z)$
  - if the sign of $f(x,y,z)$ changes from one step to the next
    → the ray intersects the surface between these two steps



high precision required → very small step → high computing time

Advanced Rendering

Global illumination problem

- *compute the exact light intensity over surfaces of the scene*

  → no approximation (such as constant ambient light)

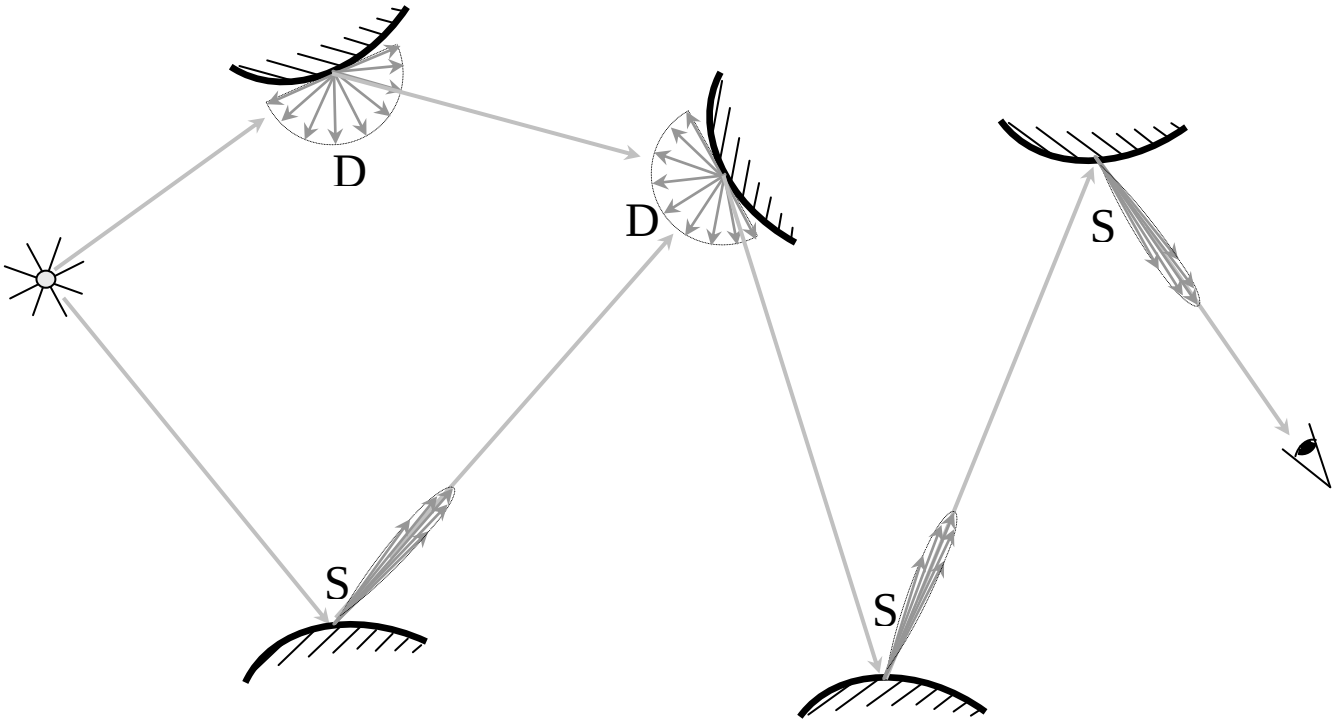
  → take into account precisely all the optical phenomena and interactions between them

- *two main classes of optical phenomena:*

  - specular reflection or refraction: light travels through a well defined path
    → easy to follow the propagation of information

  - diffuse reflection: light is scattered in every direction
    → a lot more information spread out all over space
    → much more difficult to compute propagation of this information

- light goes from the light source to a pixel of the virtual camera
  through a succession of specular and diffuse reflections:



in practice, partial solutions corresponding to particular cases for the light path:

- sequence of specular reflections only

  → *standard ray tracing* with *Phong model* for diffuse reflections

- sequence of specular reflections in which one diffuse reflection is inserted

  → *backward ray tracing* followed by *standard ray tracing*

- series of only diffuse reflections

  → *radiosity*

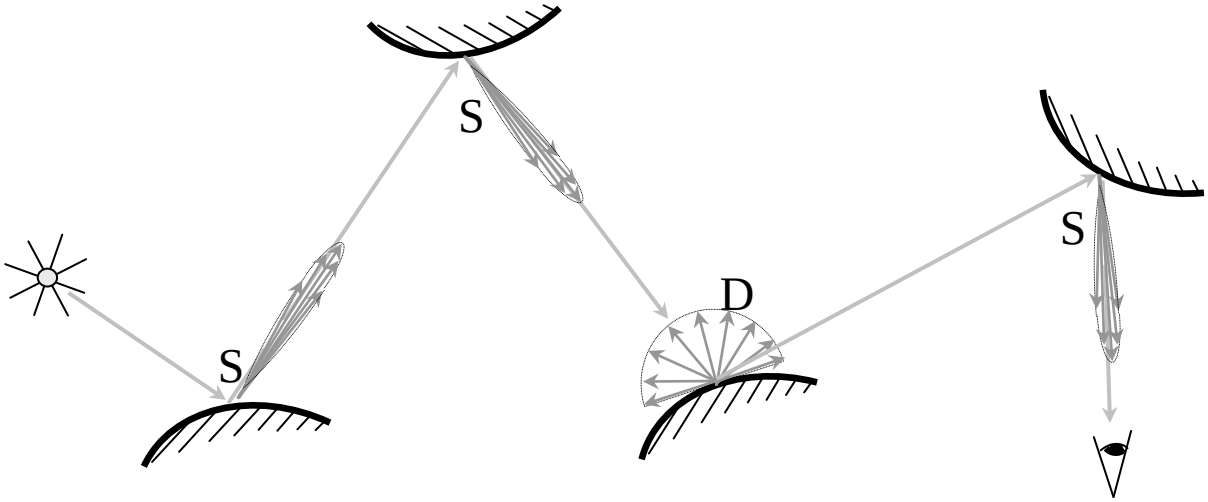
difficult to combine these partial solutions

Backward ray tracing

- *reverse direction of standard ray tracing* = we directly follow the photons

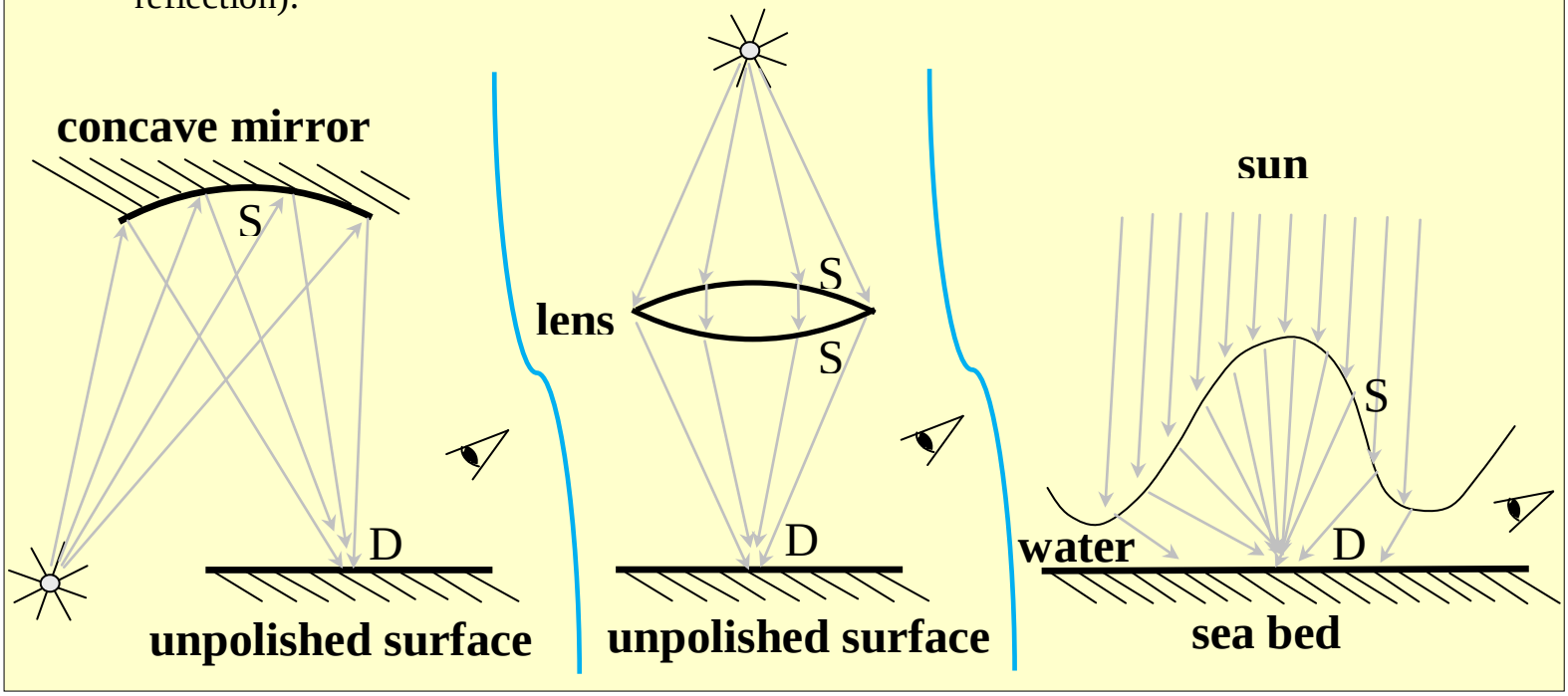
- *used in the following case:*

  light path from light source to pixel = *sequence of specular reflections*
  followed by a *diffuse reflection*
  followed by a *sequence of specular reflections*



example: *caustics*

a specular reflection / refraction highly concentrates light locally on an unpolished surface
→ on this surface, patterns of concentrated light visible from all directions (through diffuse reflection):



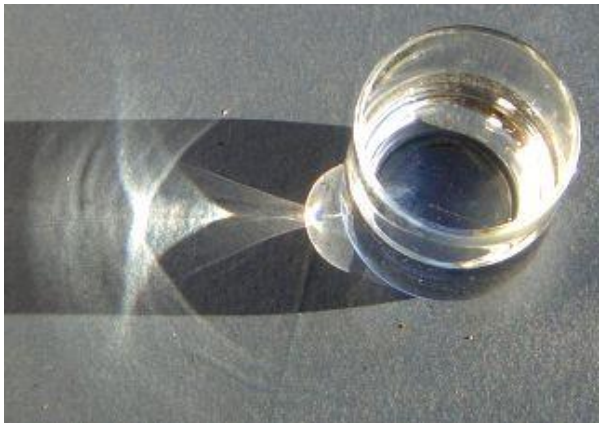


Figure courtesy
of Heiner Otterstedt, Wikipedia

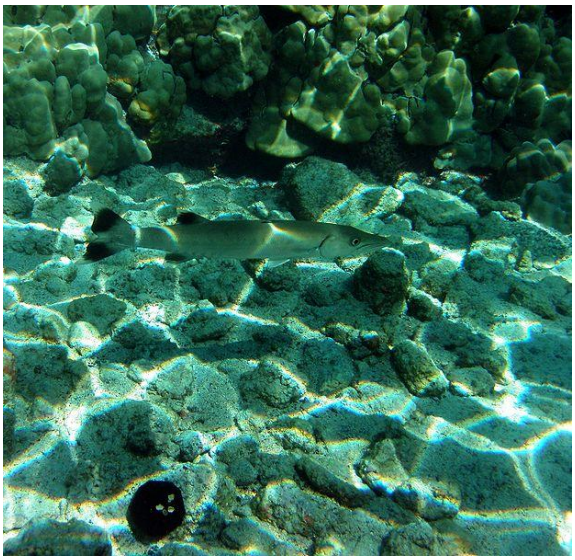


Figure courtesy
of "Mbz1", Wikipedia

the well defined path of light is “broken” at the level of the diffuse reflection

→ solve with two passes:

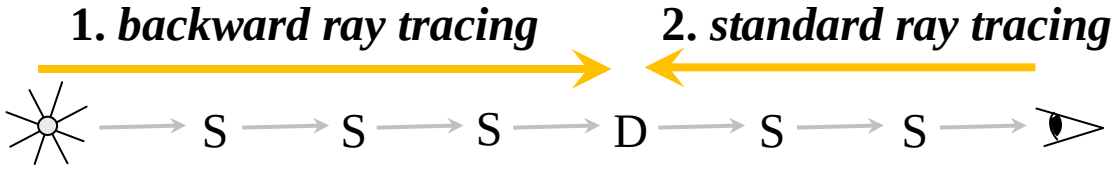
1. from the light source, propagate light forward with *backward ray tracing*

   → we obtain –and store- light intensity on the unpolished surface

2. from a given pixel, propagate light backward with *standard ray tracing*

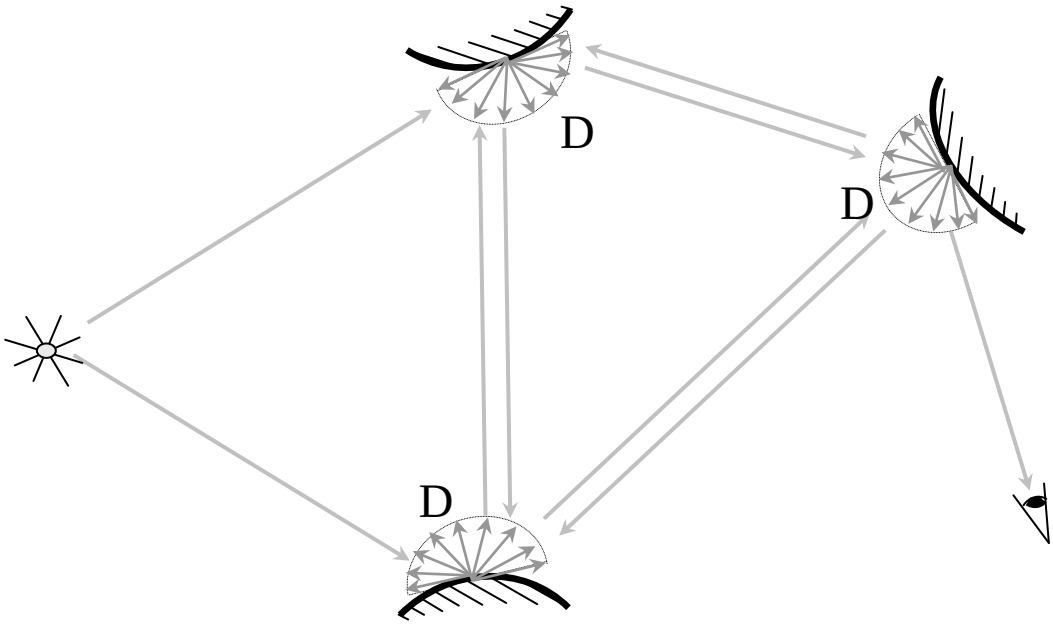
   → when we reach the unpolished surface, we use as diffuse light at this surface the value previously computed with backward ray tracing (interpolated value in fact)

= the two propagations in opposite directions along the light path join at the level of the diffuse reflection:



Radiosity

- light path from light source to pixel = *sequence of diffuse reflections only*



light bounces back and forth between all surfaces → strong interactions between them

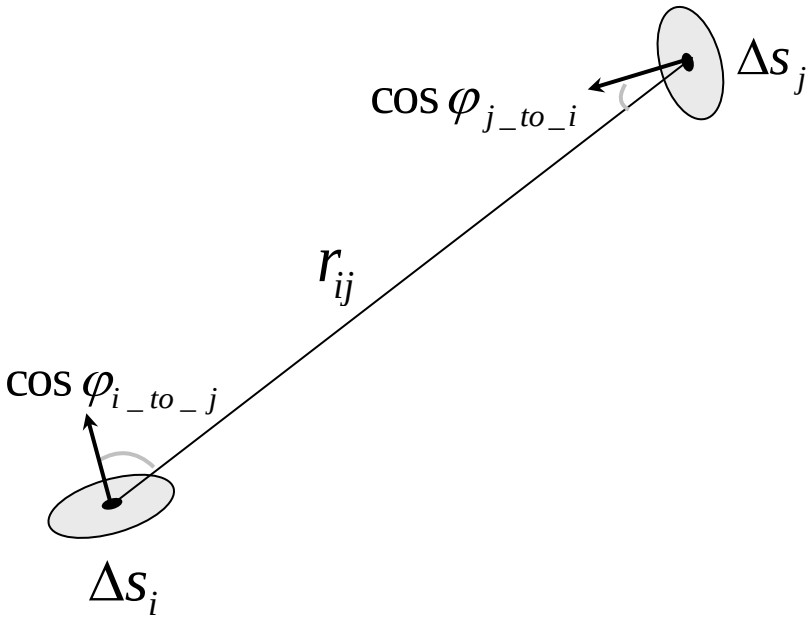
mathematical modeling of interactions between surfaces:

subdivide surfaces into many small plane surfaces

→ system of linear equations, one for each elementary surface

solving this system yields the light intensity at each elementary surface

- two interacting elementary surfaces, with orientation relative to each other:



- V_{ij} = { 1 if elementary surfaces Δs_i and Δs_j are visible from each other; 0 else

→ trace a ray between Δs_i and Δs_j and check for any intersection in between

- $\rho_i \leq 1$ reflectivity of elementary surface Δs_i

- the light intensity emitted by the elementary surface Δs_i is the sum of the light intensity I_i^E that it eventually emits by itself (if it is a light source) and of the diffuse reflection of the light rays coming directly from all other visible elementary surfaces Δs_j :

$$I_i = I_i^E + \rho_i \cdot \sum_{\substack{j \\ j \neq i}} \left(\frac{V_{ij} \cdot \cos\varphi_{i\_to\_j} \cdot \cos\varphi_{j\_to\_i} \cdot \Delta s_j}{\pi \cdot r_{ij}^2} \cdot I_j \right)$$

→ assuming the geometry of the scene is constant:

$$I_i = I_i^E + \rho_i \cdot \sum_{\substack{j \\ j \neq i}} \left(F_{ij} \cdot I_j \right)$$

F_{ij} is the *form factor*

→ *n* linear equations for the *n* elementary surfaces:

$$\begin{bmatrix} 1 & -\rho_1 \cdot F_{12} & \dots & -\rho_1 \cdot F_{1n} \\ -\rho_2 \cdot F_{21} & 1 & \dots & -\rho_2 \cdot F_{2n} \\ \dots & \dots & \dots & \dots \\ -\rho_n \cdot F_{n1} & -\rho_n \cdot F_{n2} & \dots & 1 \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \\ \dots \\ I_n \end{bmatrix} = \begin{bmatrix} I_1^E \\ I_2^E \\ \dots \\ I_n^E \end{bmatrix}$$

→ solving this linear system the light intensity at each elementary surface

we can take into account non punctual light sources
spread over several elementary surfaces (I_i^E is non null for these surfaces)
however, in practice, most of the I_i^E are null

Computing the matrix is extremely costly (rays traced between all elementary surfaces)
Solving the linear system is extremely costly as well

→ iterative resolution (*progressive radiosity*)

Figure courtesy
of David Bařina, Kamil Dudka, Jakub Filák, Lukáš Hefka, Wikipedia

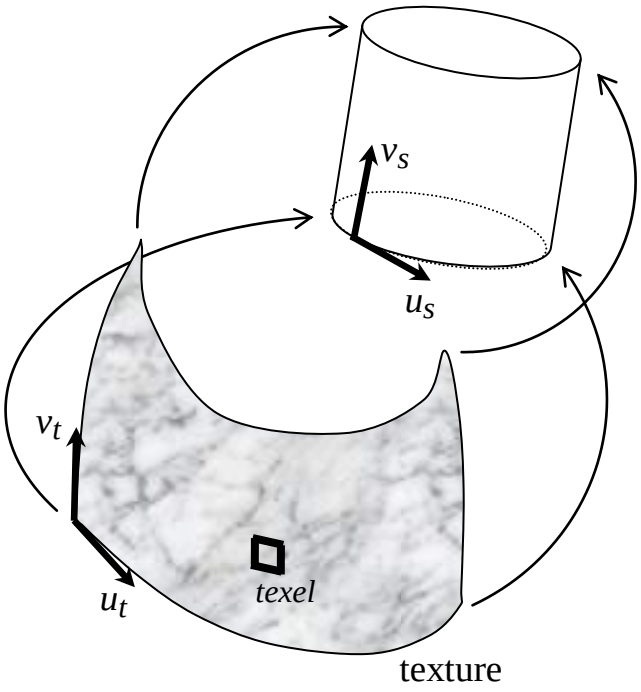
Textures

2D textures, mapping, aliasing, anti-aliasing

principle:

- *2D texture:* image which is mapped on the surface of an object

  → enhances the aspect of the object and makes it look more realistic



- *texel:* basic element of a texture (*pixel:* basic element of a picture)

- texture → *texture coordinate system* (u_t , v_t)
- parametric surface → *surface coordinate system* (u_s , v_s)

mapping:

- projection of the texture onto the surface

→ transformation from surface coordinate system into texture coordinate system:

$$\begin{cases} u_t = map_u(u_s, v_s) \\ v_t = map_v(u_s, v_s) \end{cases}$$

linear mapping:

$$\begin{cases} u_t = a_u \cdot u_s + b_u \\ v_t = a_v \cdot v_s + b_v \end{cases}$$

problem: eventual deformation of the surface

= may affect both shapes and relative sizes of texture details

mapping on a plane: no modification of shapes and relative sizes with linear mapping

mapping on a sphere: texels shrink toward the poles with linear mapping (cartography...)

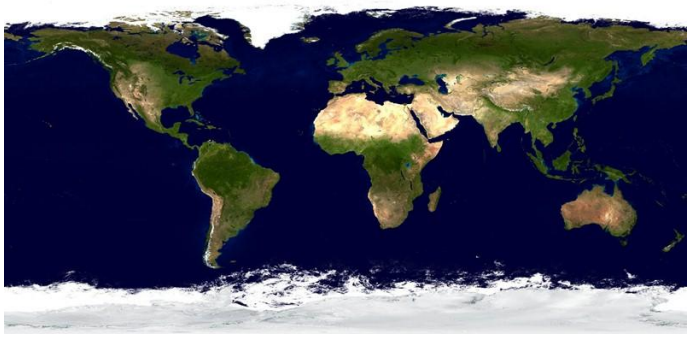
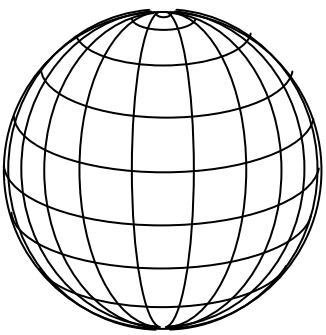
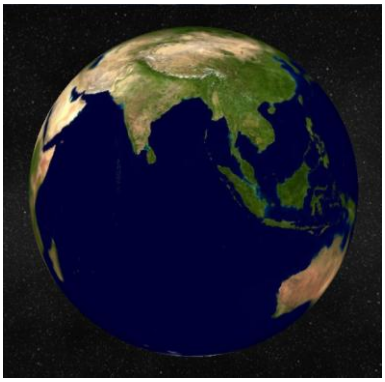


Figure courtesy of http://www.oera.net/How2/TextureMaps2.htm





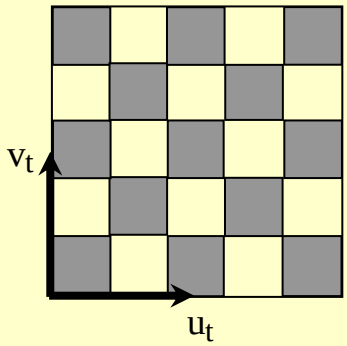
how to define a 2D texture:

- *predefined image*: photograph, map, ...

- *procedural approach* - *explicit function*:

example:

if int(u_t / Δ) + int(v_t / Δ) is even: color ← black
else: color ← white

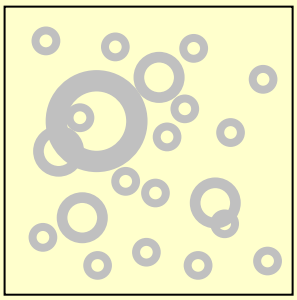


- *procedural approach* - *fractal description*: similar features repeated at various scales

example: lunar landscape (used for perturbation of normal - Bump mapping)

craters of diverse sizes are laid at random
more and more craters as they get smaller

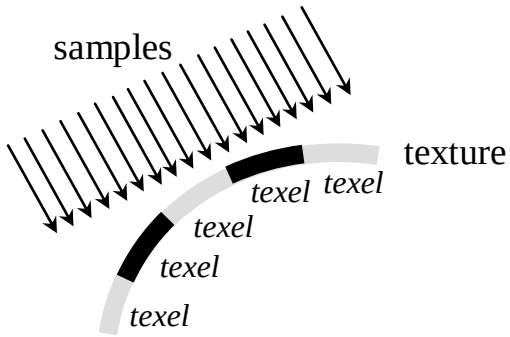
1 crater of size *h*
4 craters of size *h* / 2
16 craters of size *h* / 4
. . .



matching texels with pixels:

the projection on the screen of a texel should have roughly the same size as a pixel

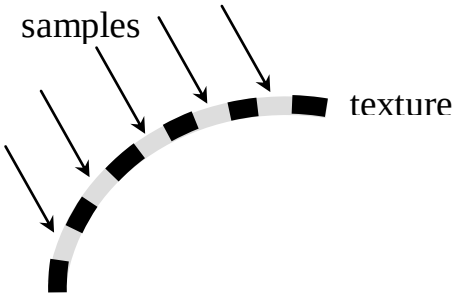
- *oversampling:* if we look too closely, the shape of the texels becomes visible:



solution to oversampling: interpolate values within each texel
→ instead of showing the texels' shape,
the texture appears smooth, blurry

- *undersampling* → *aliasing:*

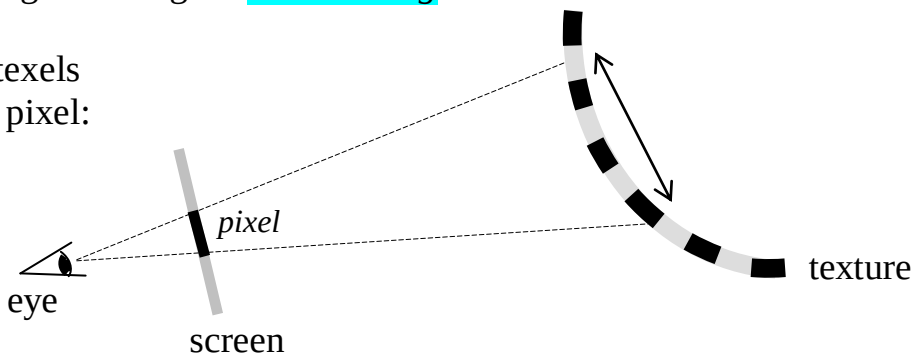
if we do not sample densely enough, we lose some details between the sampling points:



→ random artifacts appear = *aliasing*

solution to undersampling, aliasing: filtering = *anti-aliasing*

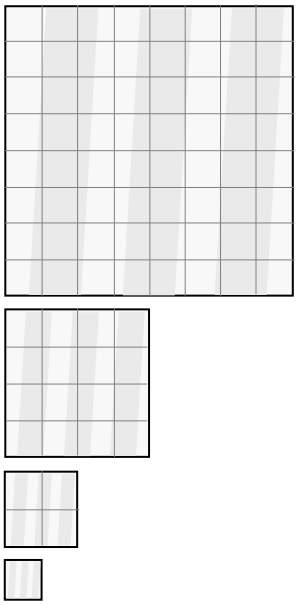
for one given pixel, average the texels
whose projection lies within this pixel:



for greater speed, precomputed filtered values:
the averages of groups of texels are stored in memory
→ MIPMAP, Summed Area Table (SAT), …

MIP MAP:

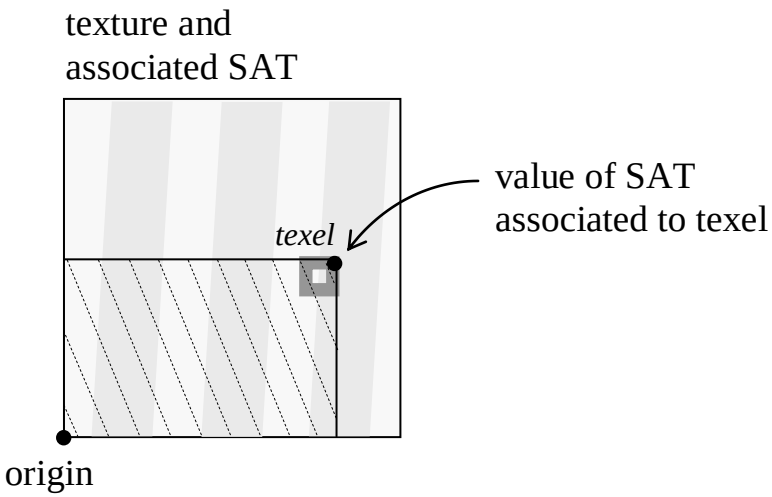
for each scale $\frac{H}{2^k}$, we store the averaged texture:



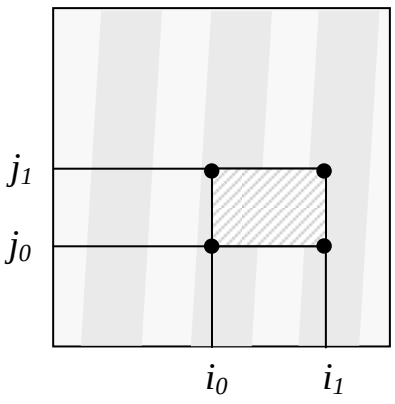
→ depending on the scale, read average in one of the arrays
(in practice, interpolation between two arrays)

Summed Area Table (SAT):

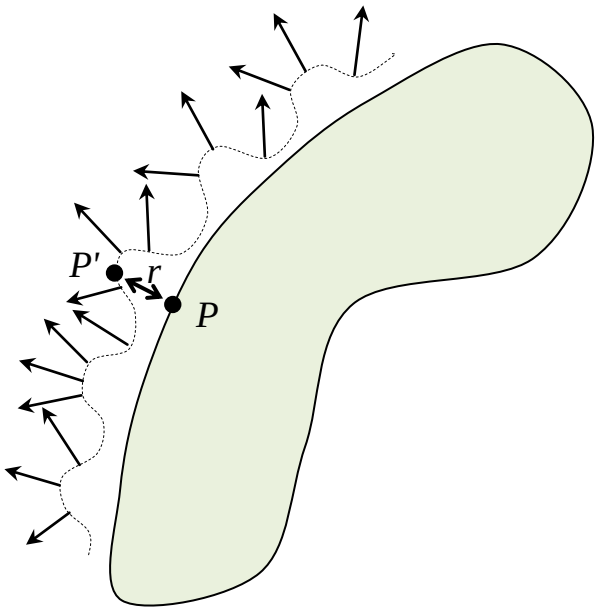
for each texel, precompute the sum of texels from the origin to this texel:



→ texture average over area $[i_0, i_1] \times [j_0, j_1] = \frac{sat[i_1, j_1] - sat[i_1, j_0] - sat[i_0, j_1] + sat[i_0, j_0]}{(i_1 - i_0) \times (j_1 - j_0)}$



Bump mapping:



small perturbations imposed to the surface normal

the surface itself is not modified

pseudo-surface = actual surface
+ small *pseudo-relief*

pseudo-relief = displacement *r* orthogonal to surface:

$$P'(u_s, v_s) = P(u_s, v_s) + r(u_s, v_s) \cdot N$$

→ tangent and normal vectors of actual surface and fake surface:

$$\begin{cases} U = \dfrac{\partial P}{\partial u_s} \\ V = \dfrac{\partial P}{\partial v_s} \\ N = U \times V \end{cases} \qquad \begin{cases} U' = \dfrac{\partial P}{\partial u_s} + \dfrac{\partial r}{\partial u_s} \cdot N \\ V' = \dfrac{\partial P}{\partial v_s} + \dfrac{\partial r}{\partial v_s} \cdot N \\ N' = U' \times V' \end{cases}$$

→ normal to fake surface: $N' = N + \dfrac{\partial P}{\partial u_s} \times \left(\dfrac{\partial r}{\partial v_s} \cdot N \right) - \dfrac{\partial P}{\partial v_s} \times \left(\dfrac{\partial r}{\partial u_s} \cdot N \right)$

partial derivatives of *r*: $\begin{cases} \dfrac{\partial r}{\partial u_s} = \dfrac{\partial r}{\partial u_t} \cdot \dfrac{\partial u_t}{\partial u_s} + \dfrac{\partial r}{\partial v_t} \cdot \dfrac{\partial v_t}{\partial u_s} \\ \dfrac{\partial r}{\partial v_s} = \dfrac{\partial r}{\partial u_t} \cdot \dfrac{\partial u_t}{\partial v_s} + \dfrac{\partial r}{\partial v_t} \cdot \dfrac{\partial v_t}{\partial v_s} \end{cases}$

→ instead of *r*, we store two texture values: $\dfrac{\partial r}{\partial u_t}$; $\dfrac{\partial r}{\partial v_t}$

3D textures

3D texture $T(x,y,z)$ = number function of position in space

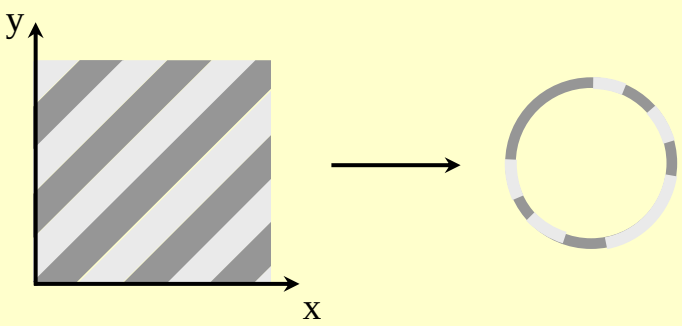
a 3D texture depends only on the position in space and NOT on the objects'geometry

- *solid texture:*

$T(x,y,z)$ may define an optical parameter (color, …) → sample $T(x,y,z)$ over the surface

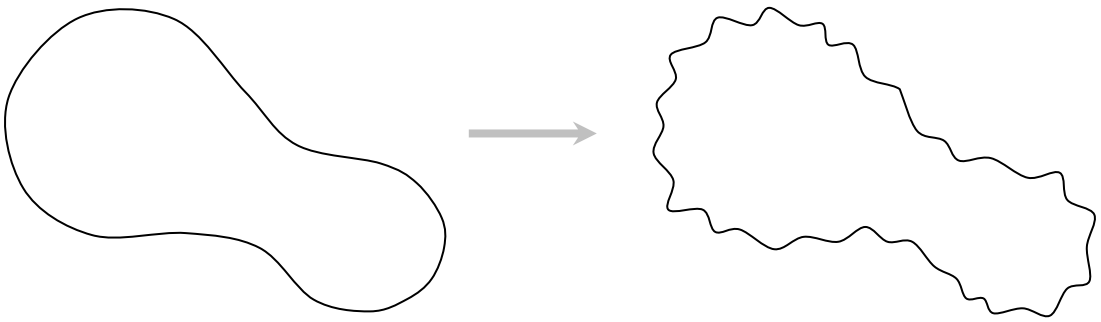
example:

if int(x + y + z) is odd: color ← black
else: color ← white



- *perturbation of an implicit surface with a 3D texture:*

$f(x,y,z) = 0$ → $f(x,y,z) + T(x,y,z) = 0$



→ can be used to model hair, or fur, stemming from a head…

Procedural modeling

Fractal landscape

- *1D perturbed subdivision:*

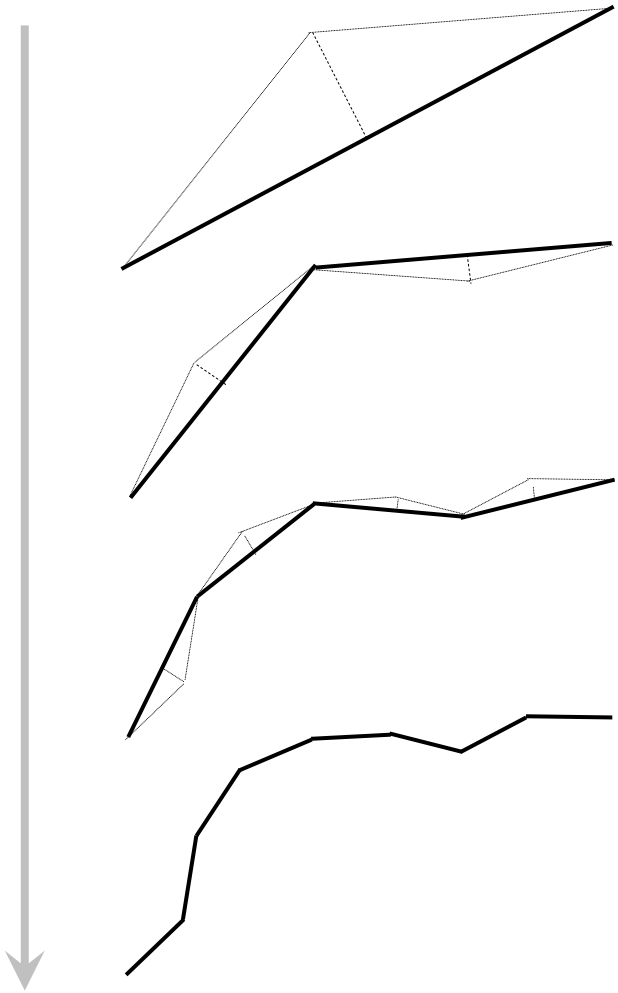
subdivide a segment into two smaller segments

→ random displacement of the middle of the segment:

$$\delta = roughness \cdot random \cdot h \cdot N \quad \text{with} \quad random \in [-1,+1]$$

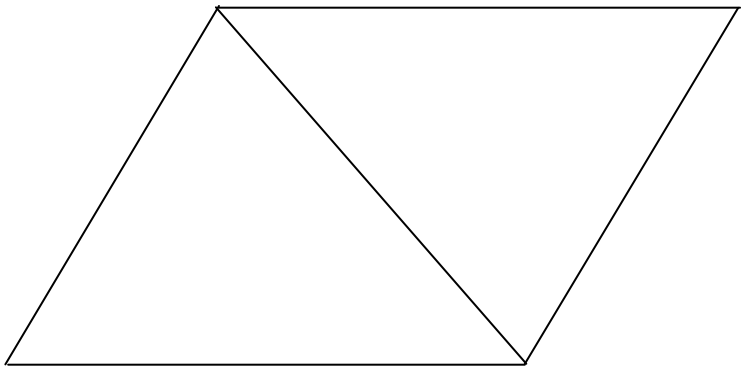
h size of the segment and N unit normal to the segment

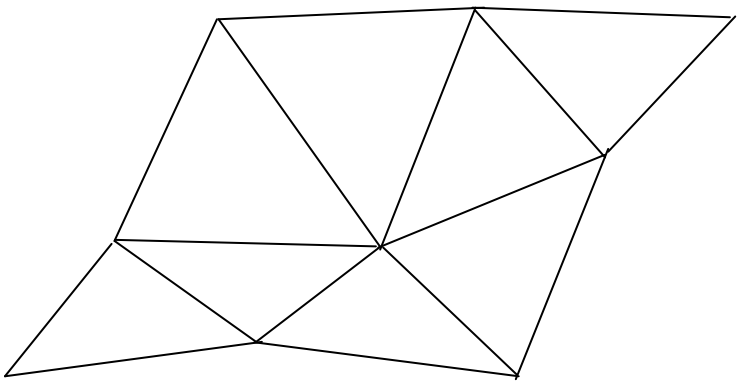
reapply recursively this subdivision on each of the smaller segments:

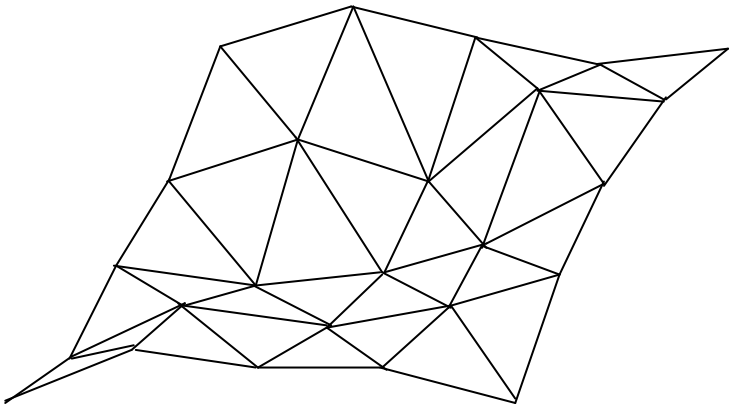


- *2D perturbed subdivision:*

similar recursive subdivision of a plane with triangles:







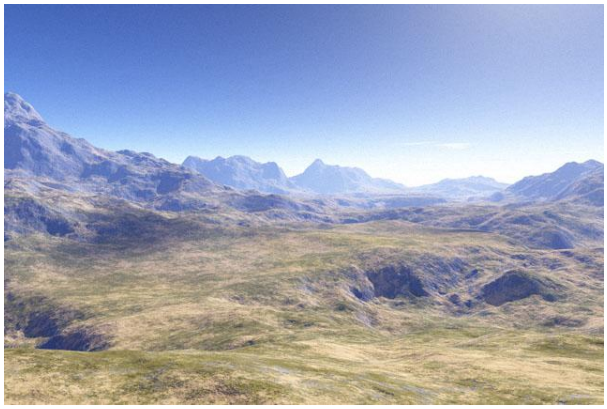


Figure courtesy of "The Ostrich", Wikipedia

Figure courtesy of "Stevo-88", Wikipedia

recursive subdivision of a quadrangle into 4 sub quadrangles → *diamond-square algorithm*

Animation

Principle - degrees of freedom of a scene

we must describe the evolution over time of all the degrees of freedom of the scene

- *degrees of freedom* = independent angles and position coordinates:
  - position and orientation of camera (3 coordinates + 3 angles)
  - position and orientation of a solid object (3 coordinates + 3 angles)
  - angles of an articulated structure

the total number of degrees of freedom may be huge (several thousand or more)

→ how to provide relevant values over time for all these degrees of freedom???

Kinematic animation

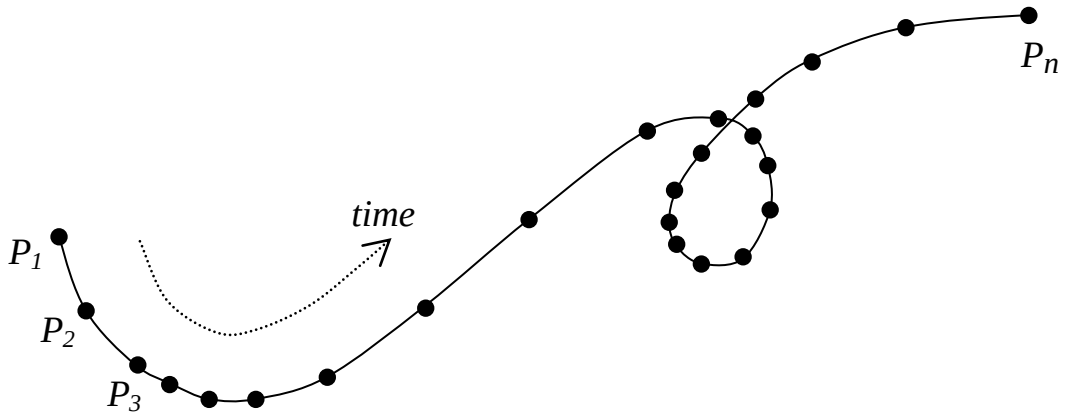
the degrees of freedom are directly given as explicit functions of time

= purely geometric description of the evolution over time

- *spline-driven animation*:

over time, an object passes through positions P_1 , P_2 , … , P_n at times t_1 , t_2 , … , t_n

→ coordinates x , y , z are interpolated over these positions with *cubic spline functions*:



Dynamic animation

simulation of a physical system:

- model *forces* and *torques* that are applied on the virtual objects
- *laws of physics* to reconstitute the evolution of the degrees of freedom

→ *numerical time integration* of the degrees of freedom (with *Runge Kutta method*)

Animation of articulated structures

kinematic animation:

- *forward kinematics*:

the values over time of the articulation angles are known explicitly

- *inverse kinematics*:

we impose the position of a part of the structure over time

(hand of a human body following, and trying to catch, a moving object)

→ we must find appropriate values over time for the angles
so that the part complies with the imposed position over time

dynamic animation:

- torques imposed on articulations (action of muscles)
- forces imposed on some parts (reaction forces in case of contact with an object)

→ through numerical time integration, we deduce the articulation angles over time

- *feedback control loop*