Sampling

- undersampling → information lost
- ↳ cannot reconstruct, as multiple solutions → "aliasing"
- „Nyquist-Frequency": $\frac{1}{2}$ sampling frequency ⇒ max frequency / bandwidth of original Signal must be smaller
- ⇒ sample twice the highest frequency found in orig. Signal

Quantization

- real value $\xrightarrow{\text{to}}$ integer value
- ↳ lossy!, use nearest integer value as new functionvalue. Sampling without Quantization is not lossy

Image Properties

- Resolution: Pixel × Pixel
- Geometric Resolution: Pixel per Area
- Radiometric Resolution: Bits per Pixel

Image Noise

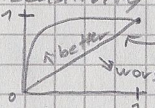
- Additive Noise: $I(x,y) = f(x,y) + c$
- → additive Gaussian: $c \sim N(\mu, \sigma^2)$
  - $\Rightarrow p(c) = (2\pi\sigma^2)^{-1} \cdot \exp(-(c-\mu)^2/2\sigma^2)$
  - ⇒ mostly $\mu = 0$!
- → Poisson noise (shot noise): $p(c) = \frac{\lambda^c e^{-\lambda}}{c!}$
- → Rician noise (MRI): $p(I) = I/\sigma^2 \exp\left(\frac{-(I^2+f^2)}{2\sigma^2}\right) I_0\left(\frac{If}{\sigma^2}\right)$
- Multiplicative Noise: $I(x,y) = I(x,y) + I(x,y) \cdot c$
- → Quantization error
- → "salt-and-pepper"
- Signal to noise ratio (SNR): $s = F/\sigma$, $F = \frac{1}{x \cdot y} \sum_{i=1}^{x} \sum_{j=1}^{y} f(x,y)$ (σ: noise variance)
- Peak SNR (PSNR): $s_p = F_{max}/\sigma$

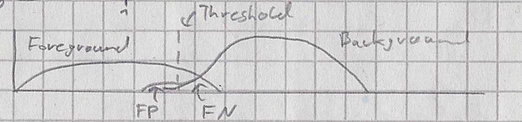
Image Segmentation

- Find segments R_i in Image I: $I = \bigcup R_i$, $R_i \cap R_j = \emptyset \; \forall i \neq j$
- Thresholding: produces binary Image $B(x,y) = 1$ if $I(x,y) \geq T$, else $B(x,y) = 0$; T is Threshold value
- Chromakeying (green/blue/...-screen)
  - Alphamask $I_\alpha := |I - g| > T$
  - e.g. green screen: $g = (0\ 255\ 0)$, $T \approx 20$
  - $\rightarrow I = I_\alpha I_a + (1 - I_\alpha) I_b$ (Foreground I_a, Background I_b)
  - ↳ Gauss verwenden, da nicht perfekt Grün haben werden
- Intensity: $i = R + G + B \rightarrow$ Normalized color: $(r,g,b) = \left(\frac{R}{i}, \frac{G}{i}, \frac{B}{i}\right)$
- I_α should be grayscale, not binary: motion blur

ROC Analysis

- "Receiver Operating Characteristics"
- TP: true positive, TN: true negative, FP: false positive, FN: false negative
- $P := TP + FN$ • $N := TN + FP$
- ROC Plot: y: TP fraction, x: FP fraction
- ↳ TP fraction: $\frac{\#TP}{P}$ (sensitivity) • FP fraction: $\frac{\#FP}{N}$ (1-specificity)





- V_{TN}: value TN, • V_{TP}: value TP, C_{FN}: cost FN, • C_{FP}: cost FP
- → choose point in ROC curve with gradient $\beta = N/P \cdot (V_{TN} + C_{FP}) / (V_{TP} + C_{FN})$
- (most of the time: $V_{TN} = V_{TP} = 0$)

Distance Measures

- $I_\alpha = |I - I_{bg}| > T$, I_{bg}: Background Image
- $T \mathrel{\hat=} (20, 20, 10)$ (example)
- ⇒ Better: $I_\alpha = \sqrt{(I - I_{bg})^T \Sigma^{-1} (I - I_{bg})} > T$
- Σ: background pixel appearance covariance matrix. Done per pixel! Use Gaussian

Morphological Operators (Binary Images)

- 8-neighbor erode ("Minkowsky Subtraction")
  - → remove (make background) foreground pixel if it has 8-neighbor background pixel
- 8-neighbor dilate ("Minkowsky addition")
  - → add (make foreground) background pixel if it has 8-neighbor foreground pixel.
- ⇒ for smoothing, remove noise/artefacts

Structuring Elements

- has origin • as array: $\{(1,1), (2,2)\}$
  - ↳ at 1,1 its 1 and at 2,2.
- S fits I at x if $\{y : y = x + s, s \in S\} \subset I$
- S hits I at x if $\{y : y = x - s, s \in S\} \cap I \neq \emptyset$
- S misses I at x if $\{y : y = x - s, s \in S\} \cap I = \emptyset$
- → fit: for every pixel set to 1 in Structuring Element, the corresponding Image pixel is also 1.
- → hits: at least one pixel set to 1 "fits" image
- ⇒ "0" ≙ "ignore" / "don't care"

- Erosion: $E = I \ominus S$, I: image, S: Str. Ele.
  - ↳ if S fits at x (x and origin overlap), set $x = 1$
- Dilation: $D = I \oplus S$: set $x = 1$, if S hits I at x
- Opening: $I \circ S = (I \ominus S) \oplus S$
- Closing: $I \bullet S = (I \oplus S) \ominus S$
- → opening: open gaps that are thin
- → closing: close holes by maintaining original dimensions
- Hit-and-Miss: $H = I \otimes S$: set $x = 1$ (origin) if 1's and 0's (!) fit exactly into the image
- thinning: $I \oslash S = I \setminus (I \otimes S)$
- thickening: $I \odot S = I \cup (I \otimes S)$
- $(I \odot S)^c = I^c \oslash S$
- $I \boxed{?} \{S_i\}_{i=1}^{n} = ((\ldots(I \boxed{?} S_1) \boxed{?} S_2) \ldots) \boxed{?} S_n)$
- Skeletonization:
  - Structuring Element

0	1	0
1	1	1
0	1	0

  - n-th Skeleton Subset: $S_n(X) = (X \ominus_n B) \setminus [(X \ominus_n B) \circ B]$ (← do n times)
  - Skeleton: $S(X) = \bigcup_{n=1}^{\infty} S_n(X)$
  - Reconstruction: $X = \bigcup_{i=1}^{\infty} S_i(X) \oplus_i B$

Linear Shift-Invariant Filtering

- → modify pixels based on neighbors
- → linear combination of neighbors
- → do the same for each pixel
- $I'(x,y) = \sum_{(i,j) \in N(m,n)} k(x,y;i,j) I(i,j)$ — neighborhood of m,n; k: kernel used
- shift invariant (same for every pixel) $\Leftrightarrow k(x,y;i,j) = k(i,j) \; \forall x,y$
- Correlation: $I' = k \circ I$
  - $\Leftrightarrow I'(x,y) = \sum_{(i,j)} k(i,j) I(x+i, y+j)$
- Convolution: $I' = k * I$
  - $\Leftrightarrow I'(x,y) = \sum_{(i,j)} k(i,j) I(x-i, y-j)$
  - → same as correlation, kernel is reversed
- → if $k(i,j) = k(-i,-j) \Leftrightarrow$ Correlation == Convolution
- kernel ⟼

k(-1,-1)	k(0,-1)	k(1,-1)
k(-1,0)	k(0,0)	k(1,0)
k(-1,1)	k(0,1)	k(1,1)

- $k(i,j)$ all $\frac{1}{9}$ (9 Fields in k): Blur with Boxfilter • $k(i,j) = -1/9$, $k(0,0) = +17/9$ is sharpening filter
- Seperable Kernels: $k(m,n) = f(m) g(n)$
  - $I'(x,y) = k(x,y) * I = f * \underbrace{(g * I)}_{I''}$
  - $\Rightarrow I'' = \sum_j g(j) I(m, n-j)$
  - $\Rightarrow I' = \sum_i f(i) I''(m-i, n)$

$$LoG(x,y) = -\frac{1}{\pi\sigma^2} \cdot \left(1 - \frac{x^2+y^2}{2\sigma^2}\right) \cdot \exp\left(-(x^2+y^2)/2\sigma^2\right)$$

- Gaussian Smoothing Kernel:
  - $k(x,y) = g(x,y) := \frac{1}{2\pi\sigma^2} \exp(-(x^2+y^2)/2\sigma^2)$
  - $= \frac{1}{2\pi\sigma^2} \exp\left(\frac{-x^2}{2\sigma^2}\right) \cdot \frac{1}{2\pi\sigma^2} \exp\left(\frac{-y^2}{2\sigma^2}\right)$
  - $= g(x) \cdot g(y)$ (seperable)!
- Differential Filters:
  - Prewitt operator: $\begin{vmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{vmatrix}$
  - Sobel operator: $\begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$
- High-pass filters:
  - Laplacian operator: $\begin{vmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{vmatrix}$
  - High-pass filter: $\begin{vmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{vmatrix}$ ← "8" ist rot
- Filters look like the effect they are intended to find
- Image sharpening: $I' = I + \alpha |k * I|$, $\alpha \in [0,1]$, k: high-pass filter

Template Matching

- search for best match by minimizing mean-squared error
  - $E(p,q) = \sum_x^{\infty} \sum_y^{\infty} (s(x,y) - t(x-p, y-q))^2$
  - $= \sum_x \sum_y |s(x,y)|^2 + \sum_x \sum_y |t(x,y)|^2 - 2 \cdot \sum_x \sum_y s(x,y) \cdot t(x-p, y-q)$
- or equivalently maximize area correlation $r(p,q) = \sum_x^{\infty} \sum_y^{\infty} s(x,y) \cdot t(x-p, y-q)$
  - $= s(p,q) * t(-p,-q)$
  - ⇒ area equivalence is convolution of image s and t.
- $s(x,y)$: original image
- $t(x,y)$: template (smaller than s)
- because of Cauchy-Schwarz: $r(p,q) \leq \sqrt{\left(\sum_x \sum_y |s(x,y)|^2\right) \cdot \left(\sum_x \sum_y |t(x,y)|^2\right)}$
  - ⇒ only equal iff: $s(x,y) = \alpha \cdot t(x-p, y-q)$ with $\alpha \geq 0$
- after calculating $E(p,q)$ or $r(p,q)$ search peaks ⇒ matches!
- remove mean before doing the matching as bright areas would attract high values (peaks)

Edge Detector

- Idea: detect local Gradient: $|grad(f(x,y))| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$
- → for digital images use kernels:
  - "difference": $[-1\ 1]$
  - "central difference": $[-1\ [0]\ 1]$
  - see above: "Prewitt" / "Sobel"
  - Roberts: $\begin{vmatrix} [0] & 1 \\ -1 & 0 \end{vmatrix}$ $\begin{vmatrix} [1] & 0 \\ 0 & -1 \end{vmatrix}$
  - if using Laplacian structure → Blur first as it equally responds to weak and strong edges.
  - ↳ Blur with Gauss results in LoG („Laplacian of Gauss") operator ←

Canny Edge detector

1) Smooth with Gaussian filter (Blur)
2) Compute gradient magnitude (thick edge length):

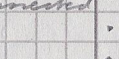
$$M(x,y) = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

3) Compute edge direction (angle) (i.e. Sobel / Prewitt Filter (rotate once (transpose) for x and y direction):

$$\alpha(x,y) = \tan^{-1}\left(\frac{\partial f}{\partial x} \Big/ \frac{\partial f}{\partial y}\right)$$

4) non-max. suppression: α rounded to 0°/180°, 45°/225°, 90°/270°, 135°/315°. If neighbors of point in α direction higher value → set to 0, otherwise keep
5) Double-thresholding of gradient magnitude
Strong edge if $M(x,y) \geq \Theta_{high}$
Weak edge if $\Theta_{high} > M(x,y) \geq \Theta_{low}$
→ $\Theta_{high}/\Theta_{weak} = 2\ldots3$
6) Reject weak edge not connected to strong edge.

Hough transform

- First: Line detection: $y = m \cdot x + b$ (or with polar coordinates (to solve the problem for $m = ?$ if vertical line) $y = \left(-\frac{\cos\alpha}{\sin\alpha}\right) \cdot x + \left(\frac{r}{\sin\alpha}\right)$ 

→ for every edge pixel (x_0, y_0) use pair $(\alpha, r) \Rightarrow r(\alpha) = x_0\cos(\alpha) + y_0\sin(\alpha)$. Now plot $r(\alpha)$ (x: α degree, y: r length). Search for points in plot with local maximal intersections → lines that are built by the points corresponding to lines in the plot that meet at local maximum. The $r(\alpha)$ plot represents all lines going through a pixel.
- Circle detection: 3D-Hough Transform with unknown (x_0, y_0, r) (first two are circle center)
→ Image preparation: Blur, use Edge detector. Hough Transform for combining / connecting
Corner Detection
- desired: accurate localization, Invariant to shift, rotation, scale, brightness changes, robust against noise
- local displacement sensitivity

$$S(\Delta x, \Delta y) = \sum_{(x,y)\in window} (f(x,y) - f(x+\Delta x, y+\Delta y))^2$$

linear approximation for small $\Delta x, \Delta y$

$$f(x+\Delta x, y+\Delta y) \approx f(x,y) + f_x(x,y)\Delta x + f_y(x,y)\Delta y$$

$$(f_x = \partial f/\partial x \quad f_y = \partial f/\partial y)$$

$$\Rightarrow S(x,y) \simeq \sum_{(x,y)\in window} \left[(f_x(x,y) \; f_y(x,y)) \begin{pmatrix}\Delta x\\ \Delta y\end{pmatrix} \right]^2$$

$$= \sum_{(x,y)\in window} \begin{pmatrix}\Delta x & \Delta y\end{pmatrix} \begin{bmatrix} f_x^2(x,y) & f_x(x,y)f_y(x,y) \\ f_x(x,y)f_y(x,y) & f_y^2(x,y)\end{bmatrix} \begin{pmatrix}\Delta x\\ \Delta y\end{pmatrix}$$

$$= (\Delta x \; \Delta y)\, M \begin{pmatrix}\Delta x\\ \Delta y\end{pmatrix} = \Delta^T M \Delta$$

M: "structure matrix" / "normal Matrix" / "second-moment Mat."

$\Rightarrow SSD \approx \Delta^T M \Delta \Rightarrow$ find points which the following is large: $\min \Delta^T M \Delta$ for $\|\Delta\| = 1 \Leftrightarrow$ maximize eigenvalues of M!
→ Keypoint detection
Look at eigenvalues of M: λ_1, λ_2.
$\Rightarrow$ Edge: $\lambda_2 \gg \lambda_1$ or $\lambda_1 \gg \lambda_2$
$\Rightarrow$ Corner: λ_1, λ_2 "large" ($k = 0.2$ … 0.5)
- "Cornerness": $C(x,y) = \det(M) - k \cdot (\text{trace}(M))^2 = \lambda_1\lambda_2 - k\cdot(\lambda_1+\lambda_2)^2$
- Better: use Gaussian weighting: $M' = \sum_{x,y\in window} G(x-x_0, y-y_0, \sigma)\, M$

↑ "Harris corner detector" ↑

- Invariant to brightness offset
- Invariant to shift / rotation
- × (not) invariant to scaling

Aliasing

Can shrink image by taking every second pixel! Typical errors resulting from aliasing:
- Wheels rolling in wrong direction
- Checkerboard misrepresented in raytrace
- Striped shirts look funny on colour TV

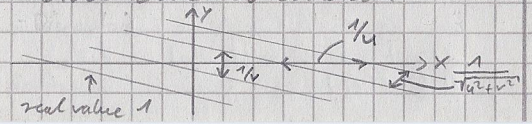
Fourier-Transformation

- Continuous: (1D ↓ 2D)
- $F(g(x))(u) = \int_{\mathbb{R}} g(x) \exp(-i2\pi ux)\, dx$
- $F(g(x,y))(u,v) = \int\int_{\mathbb{R}^2} g(x,y) \exp(-i2\pi(ux+vy))\, dx\, dy$
- Discrete ($N \times N$ Image)

$$F(g(x,y))(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} g(x,y)\, e^{-i2\pi\left(\frac{ux+vy}{N}\right)}$$

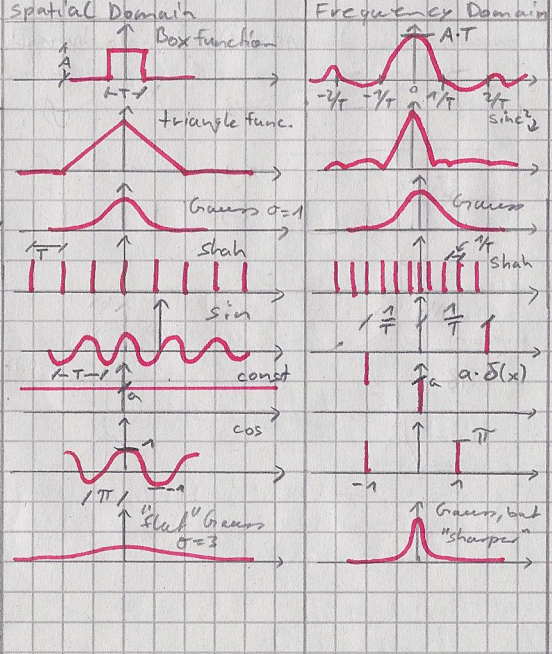
↑ transformed image

- $F = \cup f \in$ vectorized Image
↑ Fourier transform Base
- $e^{-i2\pi(ux+vy)} = \cos(2\pi(ux+vy)) - i\sin(2\pi(ux+vy))$
- Basis is const if $(ux+vy)$ is const.
- The vector (u,v) gives us:
  - Magnitude ⇔ Frequency
  - Direction ⇔ Orientation



- Real part ($\Re(F)$): How much of a cosine of that frequency you need
- Imaginary Part ($\Im(F)$): How much of a sine of that frequency you need
- Magnitude: the combined Amplitude of sine and cosine
- Phase: Relation of sine and cosine

$\Rightarrow$ Magnitude: $|F(u,v)| = \sqrt{\Re(F(u,v))^2 + \Im(F(u,v))^2}$
$\Rightarrow$ Phase: $\tan^{-1}\left(\frac{\Im(F(u,v))}{\Re(F(u,v))^2}\right)$ at (u,v)
- Natural images: very similar magnitude → Phase more interesting
- Magnitude plotted: log-scale!
→ center contains the image's mean.
→ center is $F(0,0)$
- Properties:
  - linear: $F(\alpha\cdot g(\ldots) + \beta\cdot h(\ldots)) = \alpha\cdot F(g(\ldots)) + \beta\cdot F(h(\ldots))$
  - convolution theorem:
    ~ $F(g(\ldots) * h(\ldots)) = F(g(\ldots)) \cdot F(h(\ldots))$
    ~ $F(g(\ldots)\cdot h(\ldots)) = F(g(\ldots)) * F(h(\ldots))$
    $\Rightarrow g(\ldots) * h(\ldots) = F^{-1}[F(g(\ldots))\cdot F(h(\ldots))]$
  - $x_0 \in \mathbb{R}$: $F(g(x-x_0))(u) = \exp(-i2\pi x_0 u)\cdot F(g(\ldots))$
  - $-\alpha \in \mathbb{R} \setminus \{0\}$: $F(g(\alpha x)) = \frac{1}{|\alpha|}\cdot F(g(x))\left(\frac{u}{|\alpha|}\right)$
  → scale function down ⇔ scale transfer up
  → high frequency = small detail

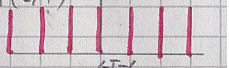


Sample + δ-function

$\delta(x) = \begin{cases}\infty & x=0\\ 0 & x\neq 0\end{cases}$ ⇒ $\int_{-\infty}^{\infty}\delta(x)\,dx = 1$

- $\delta(x) = 0$ for $x \neq 0$ and $\int_{-\infty}^{\infty}\delta(x)\,dx = 1$
- $\delta(x,y) = \delta(x)\cdot\delta(y)$
- Shah function used for creating sampling points is based on δ:

$$\text{Ш}_T(t) := \sum_{k=-\infty}^{\infty}\delta(t-k\cdot T) = \frac{1}{T}\,\text{Ш}(t/T)$$



$$= \frac{1}{T}\sum_{k=-\infty}^{\infty} e^{\exp((i2\pi kt)/T)}.$$

$\Rightarrow$ 2D Sampling of $f(x,y)$ at (x,y):

$$\text{Sample}_{2D}(f)(x,y) = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f(x,y)\cdot\delta(x-i, y-j)$$

$$= f(x,y)\cdot\left[\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i, y-j)\right]$$

- Fourier Transform of it:

$$F(\text{Sample}_{2D}(f)(x,y))(u,v)$$

$$= F\left(f(x,y)\cdot\left[\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right]\right)(u,v)$$

(convolution)

$$= F(f(x,y))(u,v) * F\left(\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right)(u,v)$$

$$= \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} F(u-i, v-j)$$

Inverse Fourier

$$f(x) = \int_{-\infty}^{\infty} F(F(x))(u)\cdot\exp(i2\pi ux)\,dx$$

(new "-" (as for other usage))

Additional information

- $\cos(x) = (e^{ix} + e^{-ix})/2$
- $\sin(x) = (e^{ix} - e^{-ix})/2$
- $e^{ix} = \cos(x) + i\sin(x)$
- $e^{-ix} = \cos(-x) + i\sin(-x) = \cos(x) - i\sin(x)$
- $\delta(x-\alpha) = \frac{1}{2\pi}\int_{\mathbb{R}} e^{ip(x-\alpha)}\,dp$
- $\delta\left(\frac{x}{2\pi} - \frac{\alpha}{2\pi}\right) = \int_{\mathbb{R}} e^{ip(x-\alpha)}\,dp$
- $2\pi\,\delta(\alpha - x) = \delta\left(\frac{x}{2\pi} - \frac{\alpha}{2\pi}\right)$

Unitary Transform

- digital Image can be viewed as a matrix:

$$f = \begin{bmatrix} f(0,0) & f(1,0) & \cdots & f(N-1,0) \\ f(0,1) & f(1,1) & \cdots & f(N-1,1) \\ \vdots & \vdots & \ddots & \\ f(0,N-1) & f(1,N-1) & \cdots & f(N-1,N-1) \end{bmatrix}$$

where $f(x,y) = f_{x,y}$ denotes the pixel value at (x,y).

- The image f with $L \times N$ pixel can also be written as a vector for easier definition of some calculations:

$$f = [f(0,0)\ f(1,0)\ f(2,0) \ldots f(N-1,0)\ f(0,1)\ f(1,1) \ldots f(N-1,1) \ldots f(N-1,L-1)]^T$$

- linear image processing:

$\vec{g} = H\vec{f}$ ($\vec{f}$: image vector)

(H: operation, not needed to be square)

- Unitary transform:
  - $\vec{c} = A\vec{f}$ ($\vec{c}$: resulting image; A: unitary $L \cdot N \times L \cdot N$ matrix; real → A^T, complex → A^H)
  - A unitary, iff: $A^{-1} = A^T = A^H$
  - if A real valued: transform is „orthonormal"
  - Energy conservation: $\|\vec{c}\|^2 = \vec{c}^H\vec{c} = \vec{f}^H A^H A \vec{f} = \|\vec{f}\|^2$
  - ⇒ A is a rotation of the coordinate system (with x/y flips maybe)
  - ⇒ vector lengths („Energie") is conserved

Image collection

- f_i one image (row vector most likely)
- $F = [f_1\ f_2 \ldots f_n]$ image collection
- $R_{ff} = E[f_i \cdot f_i^H] = \frac{f_i f_i^H}{n}$ (vector · vector^H)

Energy distribution + Unitary Transf.

- Autocorrelation of $\vec{c} = A\vec{f}$:

$R_{cc} = E[\vec{c}\vec{c}^H] = E[A\vec{f} \cdot \vec{f}^H A^H] = A R_{ff} A^H$

↳ mean squared values („average energies") of the c_i are on the diagonal of R_{cc}. $\vec{c} = [c_0, c_1, \ldots, c_{N-1}]^T$

$E[c_i^2] = [R_{cc}]_{i,i} = [A R_{ff} A^H]_{i,i}$

- Eigenmatrix Φ of autocorrelation matrix R_{ff}:
  - Φ is unitary (i.e. $U^H U = U U^H = I$)
  - columns of Φ form the eigenvectors of R_{ff}: $R_{ff}\Phi = \Phi\Lambda$ where Λ is diagonal matrix containing in the diagonal the eigenvalues $\lambda_0, \ldots, \lambda_n$
  - R_{ff} is symmetric nonnegative definite ⇒ $\forall i: \lambda_i \geq 0$
  - R_{ff} is normal matrix ⇒ $R_{ff}^H R_{ff} = R_{ff} R_{ff}^H$
  - ⇒ unitary eigenmatrix Φ exists (or „PCA")

Karhunen-Loeve Transform („KL")

- Unitary transform where $A = \Phi^H$, where Φ's columns are ordered according to decreasing eigenvalue
- Transform coefficients are pairwise uncorrelated: $R_{cc} = A R_{ff} A^H = \Phi^H R_{ff} \Phi = \Phi^H \Phi \Lambda = \Lambda$
- Energy concentration property:
  - no other unitary transform packs as much energy into the first y coefficients, where y is arbitrary
  - Mean squared approximation error by choosing only first y coefficients is minimized

Eigenimages / Basis images

- For any unitary transform, the inverse is: $\vec{f} = A^H\vec{c}$ and can be interpreted in terms of the superposition of „basis images" (columns of A^H) of size MN
- For the KL transform the basis images, which are the eigenvectors of R_{ff}, are called „eigenimages"
- If energy concentration works well, only the eigenimages of the y dimensional subspace are needed to approximate images with a small error
  ⇒ tailor KL transform for image type at hand → work only with y dimensional subspace.

Application to faces (Eigenfaces)

- „observation vector" x (column vector) contains image of the face we want to find a match for in our database
- Use 8 dimensional space of faces → we can generate faces by changing 8 coefficients
- Limitation: illumination creates great differences

PCA / KL / KLT Math

I_i: i-th image in DB, I: image to find a match I_k for.

$I_i - \bar{I} \approx \sum p_i - \sum p = \sum (p_i - p)$

$\|I_i - \bar{I}\| \propto \|p_i - p\|$

approximate $\arg\min D_i = \|I_i - I\|$ with $\arg\min \|p_i - p\|$

$p = E^T \hat{I} = E^T(I - \bar{I})$ ($\hat{I}$)

$\bar{I}$: „mean image": $\frac{1}{n}\sum_i I_i$

$\hat{I}_i = I_i - \bar{I}$: Image I_i without the „mean Image"

↑ „Eigenspace matching"

JPEG

- use 8×8 Image blocks and transform them using the discrete cosine transform (DCT)
- DCT: only real numbers, fast implementation.
- Block size: - small: fast and correlation exists between neighboring pixel
  - large blocks: better compression in smooth regions
  - Use Entropy coding („Huffman code") for optimal average code word length $H = -\sum p(s) \log_2(p(s))$

Image Pyramid

- apply filter and scale down to get new Pyramid level
- top of pyramid is the smallest image
- Each level can be used for easier: edge detection, matching
- Gaussian pyramid:
  - apply gauss, sample to get next level
  - Analysis done on top image
  - e.g. „4 pixels combined are one pixel of next level

Laplace Pyramid

- for Level i: take image at level i of Gauss pyramid and subtract upscaled image of level $i+1$ in Gauss pyramid

Optical Flow

- = apparent motion of brightness patterns. „Ideally the projection of 3D velocity vectors on the image
- $I(x,y,t)$: brightness at time t and position (x,y)
- Brightness constancy assumption:

$I(x + \frac{dx}{dt}\delta t, y + \frac{dy}{dt}\delta t, t + \delta t) = I(x,y,t)$

- Optical flow constraint equation:

$dI/dt = \partial I/\partial x \cdot dx/dt + \partial I/\partial y \cdot dy/dt + \partial I/\partial t = 0$

- The aperture („Blende") problem:

$u = dx/dt \quad v = dy/dt \quad I_x = \partial I/\partial x \quad I_y = \ldots \quad I_t = \ldots$

→ $I_x u + I_y v + I_t = 0$ (1 equation, 2 unknown)

Horn + Schunk algorithm

- additional smoothness constraint

$e_s = \iint ((u_x^2 + u_y^2) + (v_x^2 + v_y^2))\,dx\,dy$

- beside our optical flow constraint ↑
- $e_c = \iint (I_x \cdot u + I_y \cdot v + I_t)^2\,dx\,dy$

⇒ minimize $e_s + \lambda e_c$

Euler-Lagrange equations:

$F_u - \partial/\partial x F_{u_x} - \partial/\partial y F_{u_y} = 0$

$F_v + \partial/\partial x F_{v_x} - \partial/\partial y F_{v_y} = 0$

In our case: $F = (u_x^2 + u_y^2) + (v_x^2 + v_y^2) + \lambda(I_x u + I_y v + I_t)^2$ ⇒ Euler-Lagrange equations are:

$\Delta u = \lambda(I_x u + I_y v + I_t) I_x$ } where $\Delta = \partial^2/\partial x^2 + \partial^2/\partial y^2$

$\Delta v = \lambda(I_x u + I_y v + I_t) I_y$ }

- example (the algo) of regularization
- solved iteratively (PDEs)
- Errors at boundaries

Lukas-Kanade (integrate over a patch)

$E(u,v) = \sum_{x,y \in \Omega} (I_x(x,y) \cdot u + I_y(x,y) \cdot v + I_t)^2$

$\frac{dE(u,v)}{du} = \sum 2 \cdot I_x (I_x u + I_y v + I_t) = 0$

$\frac{dE(u,v)}{dv} = \sum 2 \cdot I_y (I_x u + I_y v + I_t) = 0$

solve with:

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = -\underbrace{\begin{pmatrix} \sum I_x I_t \\ \sum I_y I_t \end{pmatrix}}_{b}$$

$\underbrace{(\sum \nabla I (\nabla I)^T)}_{M} \vec{U} = -\sum \nabla I I_t$ (=)

At each pixel compute $\vec{U}$ in $M\vec{U} = b$

- M singular if all gradient vectors point in same direction
  → only normal flow available

Video compression

- Humans sensitive to motion
- Visual (Human) perception limited to < 29 Hz. Cinema: 24 Hz. TV: 25/50 Hz
- Flicker can be perceived up to 60 Hz.
- Interlaced video format: 2 (temporally) shifted Images ⇒ increase of frequency from 25 → 50 Hz (per Herz 2 Images)
- Temporal processing (compression)
  - Transform/subband methods:
    - good for constant velocity global uniform motion ⇒ bad (inefficient) for real-world motion
  - Predictive methods: better for RL
- 3 Types of frames:
  - I-frame: Intra-coded frame, coded independently of all other frames
  - P-frame: Predictively coded frame, based on previous frame
  - B-frame: Bi-directionally predicted frame, coded based on both previous and future frames
- Temporal redundancy reduction ineffective if: (i) many scene changes (ii) High motion
- Use Motion-compensated prediction („MC") for temporal processing:
  - partition frame in $N \times M$ Blocks
  - describe motion of block by finding the best matching block $f(n_1, n_2, k_{cur}) \stackrel{!}{=} f(n_1 - mv_1, n_2 - mv_2, k_{ref})$
  - ⇒ motion vector (mv_1, mv_2)
- Use a metric: $g(n_1, n_2) = f(n_1, n_2, k_{cur}) - f(n_1 - mv_1, n_2 - mv_2, k_{ref})$
  - ⇒ $MSE = \frac{1}{(n_1,n_2)} \sum\sum (g(n_1,n_2))^2$ or $MAE = \frac{1}{(n_1,n_2)} \sum\sum |g(n_1,n_2)|$
- Candidate blocks: all blocks in e.g. $(\pm 32, \pm 32)$ pixel area
- search strategies for matching blocks: (i) full search (examine all candidate blocks) (ii) partial (fast) search: examine a carefully selected subset
- motion vector: estimate of motion for best matching block
- Block matching: ⊕ good robust performance for compression ⊖ often produces blocking artefacts (fine, if used with Block DCT)

- Scalable Video Coding:
  - Temporal scalability - Spatial scalability - SNR (quality) scalab.
  - → adapts to bandwidth, computational power, etc. → defines important bits (base line) and less important

Graphics Pipeline

- CPU → Vertex Processing → Rasterization → Fragment Processing → Display

or in our case:

- CPU → Vertex Shader* → Interpolation → Fragment Shader* → Display

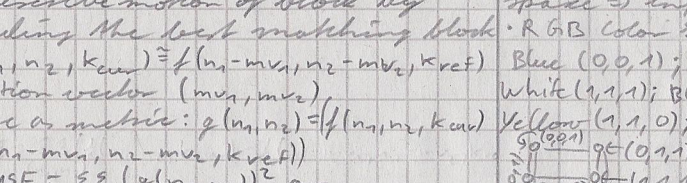
(programmable: * shaders)

- Attributes (per Vertex) → [Vertex Shader] (some code) → Varying (per Vertex) → [Interpolation] → Varying (per Fragment) → [Fragment Shad.] (some code) → Fragment color (per fragment)
- Uniforms (constants) available in both shaders!
- Fragment ≠ Pixel (store additional values: color, window id, depth, pos, ...)

Light/Color

- can be a mixture of many wavelengths
- SPD: Spectral Power distribution: $P(\lambda)$: intensity of wavelength λ
- Metamers: Two different SPD result in us in same color. Reason: infinite space projected to 3D space ⇒ information lost!
- RGB Color Space (unit cube)

Blue (0,0,1); Cyan (0,1,1); Magenta (1,0,1); White (1,1,1); Black (0,0,0); Red (1,0,0); Yellow (1,1,0); Green (0,1,0)



- color matching function as Matrix maps the testlight to RGB
- "XYZ Color Space": $X = \int_0^\infty P(\lambda)\,\bar{x}(\lambda)\,d\lambda$, $Y = \int_0^\infty P(\lambda)\,\bar{y}(\lambda)\,d\lambda$, $Z = \int_0^\infty P(\lambda)\,\bar{z}(\lambda)\,d\lambda$
- "xyz" Colorspace: "chromaticity" (x, y)
  - → use XYZ values: $x = \frac{X}{X+Y+Z}$, $y = \frac{Y}{X+Y+Z}$
  - → x, y defines color, Z the brightness
- CIE Chromaticity Chart:
  - Primary colors along curved boundary
  - linear combination of two colors: line connecting the colors
  - ... of 3 colors: triangle spanned

- CMY Color space: • for passive color systems (printers) • inverse of RGB: $[C\ M\ Y]^T = [1\ 1\ 1]^T - [R\ G\ B]^T$
- CMYK: add black as color (see Formula above for unit cube)
- YIQ: Luminance Y, In-Phase I (orange-blue), Quadrature Q (purple-green)
  - ↳ good for natural and skin colors
- HSV/HSL/HSB: user picking color spaces:
  - ↳ Hue: base color, Saturation: purity of color, value/lightness, brightness
- all color spaces above: two colors near each other in space not necessarily similar! But important concept: "Perceptually-Uniform"

Transformation

Homogeneous coord

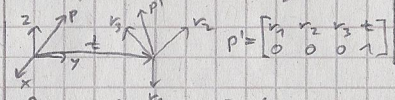
2D Translation: $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$ $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

2D Scaling: $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$ $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

2D Rotation: $\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$ $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

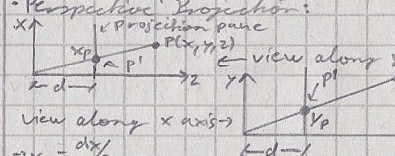
Shear along x ("verziehe"): $\begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ y: $\begin{bmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

- Combining: Matrix multiplication in reversed order.
- Change coordinate system



$P' = \begin{bmatrix} v_1 & v_2 & v_3 & t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$

- Perspective Projection:



$\Rightarrow x_p = d\,x/z$, $y_p = d\,y/z$ $\Rightarrow M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \Rightarrow p' = M_{per}\,p$

$= M_{per} \cdot p = M_{per} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \frac{z}{d} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} \frac{x\,d}{z} \\ \frac{y\,d}{z} \\ d \end{pmatrix}$ (4D, 3D!)

- Parallel: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = M_{ort}$

Open GL Transforms

Vertex $(x, y, z, 1)^T$ → Model View Transform → Eye Coordinates → Projection → Clip Coordinates → Perspective Division → Normalized Device Coord. → Viewport Transform → Window (Screen) Coord.

- Model View transform: Model to world coordinates: $\begin{bmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} m_x \\ m_y \\ m_z \\ 1 \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \\ w_z \\ 1 \end{bmatrix}$ t: translation, r: rotation

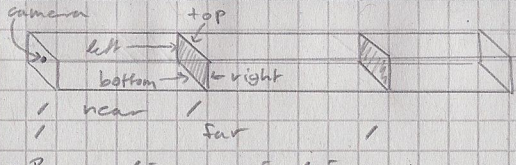
w: world coord. m: model coord.

- Model View 2nd step: World to camera coordinates: $\begin{bmatrix} l & u & -d & eye \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ c_z \\ 1 \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \\ w_z \\ 1 \end{bmatrix}$

$l = [1\ 0\ 0]^T$ $u = [0\ 1\ 0]^T$ $d = [0\ 0\ -1]^T$

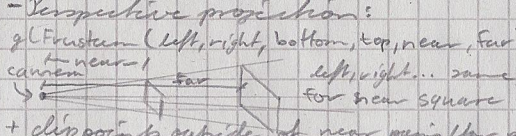
$eye = [0\ 0\ 0]^T$ c: camera coord. w: world coord.

- Projection: • Parallel (Orthogonal): gl Ortho (left, right, bottom, top, near, far)



- Perspective projection:

gl Frustum (left, right, bottom, top, near, far)



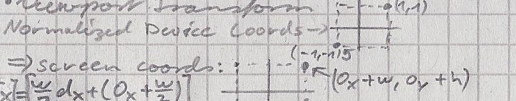
+ clip points outside of near pair/far pair

- Perspective division: d_z: depth, $d_x d_y$: screen coords.

$\begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} = \begin{bmatrix} c_x/-c_z \\ c_y/-c_z \\ c_z/-c_z \end{bmatrix}$

- Viewport Transform

Normalized Device Coords →



$\Rightarrow$ screen coords: $\begin{bmatrix} x = \frac{w}{2} d_x + (0_x + \frac{w}{2}) \\ y = \frac{w}{2} d_y + (0_y + \frac{w}{2}) \\ z = \frac{f-n}{2} d_z + \frac{f+n}{2} \end{bmatrix}$

gl Viewport (0_x, 0_y, w, h)

gl DepthRange (n, f)

Light and Shading

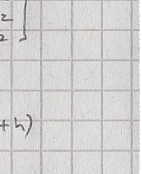
- Angle: $\theta = l/r$, circle $= 2\pi$ rad
- Solid Angle (2D angle in 3D space). sphere has 4π steradians
- Sphere: a point on the unit sphere ($r = 1$) is parameterized by two angles: $\vec{w} = (\theta, \phi)$, θ: "zenith", ϕ: "azimuth"

$\vec{w}_x = \sin\theta \cdot \cos\phi$, $\vec{w}_y = \sin\theta \cdot \sin\theta$, $\vec{w}_z = \cos\theta$

↳ θ angle between z-axes and $\vec{w}$, ϕ angle between x-axes and $\vec{w}$ projected onto xy-plane: → x, ↑ z, ↙ y

↳ latitude: $90/\pi \cdot (\pi - \theta)$

↳ longitude: $90/\pi \cdot \phi$



- Differential of solid angle:

$\rightarrow dA = (r\,d\theta)(r \cdot \sin(\theta)\,d\phi)$

$d\vec{w} = dA/r^2 = \sin\theta\,d\theta\,d\phi$

$\rightarrow \Omega = \int_{S^2} 1 \cdot d\vec{w} = \int_0^{2\pi} \int_0^{2\pi} \sin(\theta)\,d\theta\,d\phi = 4\pi$

- Light consists of: (i) x: position (ii) $\vec{w}$: direction (iii) λ: wavelength
- each photon has energy $h \cdot c/\lambda$ where h, c are const. Energy is $J = kg \cdot m^2/s^2$
- Flux $\Phi(A)$ [$J/s = W$] total energy passing through surface A per time unit • Irradiance: flux per unit area: $E(x) = d\Phi(A)/dA(x)$ [W/m^2]
- Radiosity: flux per unit area leaving surface: $B(x) = d\Phi(A)/dA(x)$ [W/m^2]
- Irradiance: $E = \Phi/A$
- Radiant intensity: Flux per solid angle: $I(\vec{w}) = d\Phi/d\vec{w}$ [W/sr], $\Phi = \int_{S^2} I(\vec{w})\,d\vec{w}$
- Radiance: Intensity per unit area $L(x, \vec{w}) = dI(\vec{w})/dA(x) = \dots = d^2\Phi(A)/(d\vec{w}\,dA(x)\cos\theta$ [$W/m^2 sr$]

Reflection Models

BRDF: Bidirectional Reflectance Distribution Function: $f_r(x, \vec{w}_i, \vec{w}_r) = dL_r(x,\vec{w}_r)/dE_i(x,\vec{w}_i) = dL_r(x,\vec{w}_r)/L_i(x,\vec{w}_i)\cos(\theta_i)\,d\vec{w}_i$ [1/sr]

↳ $\vec{w}_i$ (and other "i"): light source hitting surface. $\vec{w}_r$ (and other "r"): reflection from point of source hitting surface

Reflection Equation:

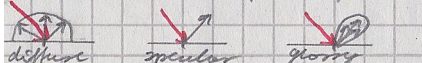
$f_r(x,\vec{w}_i,\vec{w}_r) = dL_r(x,\vec{w}_r)/L_i(x,\vec{w}_i)\cos\theta_i\,d\vec{w}_i$

$f_r(x,\vec{w}_i,\vec{w}_r)\cdot L_i(x,\vec{w}_i)\cos\theta_i = dL_r(x,\vec{w}_r)/d\vec{w}_i$

$\int_{H^2} f_r(x,\vec{w}_i,\vec{w}_r)\cdot L_i(x,\vec{w}_i)\cos\theta_i\,d\vec{w}_i = L_r(x,\vec{w}_r)$

describes a local (!) illumination model

Simpler models for reflection:



↳ For diffuse BRDF is const: $L_r(x) = f_r\,E_i(x)$

$= f_r \int_{H^2} L_i(x,\vec{w}_i)\cos\theta_i\,d\vec{w}_i = \dots$

- Ambient light: - scattered by environment, - from all directions - independent of: (i) camera pos. (ii) light position (iii) surface orient.

- reflected intensity: $I = I_a \cdot k_a$ (light source, material parameter)

- Diffuse reflection: - dependent on: (i) orientation of surface (ii) light source position - independent of camera position

$I = I_p\,k_d\cos\Theta = I_p\,k_d\,(N\cdot L)$

↳ reflected intensity; L: light source pos. N: surface normal (dot prod.)

- Simple Model: Sum of ambient light and diffuse reflection.

- Attenuation: $f_{att} = 1/d_L^2$, model the excel: $f_{att} = \min(1/(c_1 + c_2 d_L + c_3 d_L^2), 1)$.

↳ add to our simple model sum:

$I = I_a k_a + f_{att}\cdot I_p k_d (N\cdot L)$

↳ depends now also on the camera position!

- Ambient: 3D structure not visible (looks 2D), everywhere same color

→ Diffuse: 3D structure visible, also light source direction

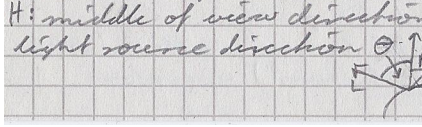
→ Specular: camera more important, darker than Diffuse

- Phong Illumination Model:

$I_\lambda = I_{a\lambda} k_a + f_{att} I_{p\lambda}\left(k_d(N\cdot L) + k_s(R\cdot V)^n\right)$

n: big n ↔ small glossy circle ($\cos\Theta$); small n ↔ big glossy circle ($\cos\alpha$)

H: middle of view direction and light source direction



- $\cos^n(\beta) = (N\cdot H)^n$ • $H = \frac{L+V}{\|L+V\|}$

- addition specular colors: $I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda}[k_d O_{d\lambda}(N\cdot L) + k_s(R\cdot V)^n]$

- Phong: the one generating a whitish circle

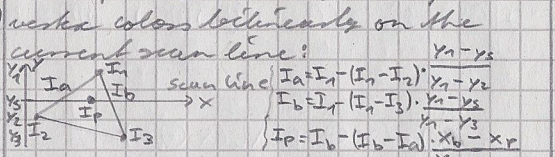
- Flat Shading: one color per primitive

- Gouraud Shading: linear interpolation of vertex intensities

- Phong Shading: linear interpolation of vertex normals

→ Gouraud Shading:

(i) calculate face normals (ii) calculate vertex normals by averaging (iii) evaluate illumination model for each vertex (iv) interpolate vertex colors bilinearly on the current scan line:



$I_a = I_1 - (I_1 - I_2)\cdot\frac{y_1 - y_a}{y_1 - y_2}$

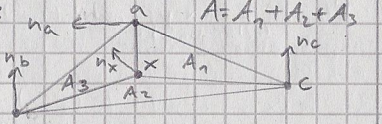
$I_b = I_1 - (I_1 - I_3)\cdot\frac{y_1 - y_a}{y_1 - y_3}$

$I_p = I_b - (I_b - I_a)\cdot\frac{x_b - x_p}{x_b - x_a}$

Problems: - perspective distortion • orientation dependence • Shared vertices

Quality depends on size of primitives

→ Phong shading: Barycentric interpolation of normals on the triangles:



$A = A_1 + A_2 + A_3$

$\lambda_a = A_2/A$

$\lambda_b = A_1/A$

$\lambda_c = A_3/A$

$n_x = \lambda_a n_a + \lambda_b n_b + \lambda_c n_c$

Properties: • $x = a \Rightarrow n_x = n_a$

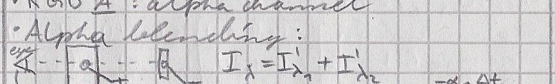
• $\lambda_a + \lambda_b + \lambda_c = 1$ • $\lambda_a\cdot a + \lambda_b\cdot b + \lambda_c\cdot c = x$

Problem: if normal not defined or representative.

Transparency

- RGB A: alpha channel

- Alpha blending:



$I_\lambda = I'_{\lambda_1} + I'_{\lambda_2}$

$I_{\lambda_2} = I_{a\lambda_1}\cdot\alpha_1\cdot\Delta t + I_{\lambda_2} e^{-\alpha_1\Delta t}$

$\Rightarrow$ linearization: $I_\lambda = I_{\lambda_1}\alpha_1 + I_{\lambda_2}(1-\alpha_1)$

because: $I'_{\lambda_2} = I_{\lambda_2}(1-\alpha_1\Delta t)$

Geometric representation

- subdivision surface, - Point set surface, - implicit surface

- Polygon Meshes:

- Boundary of objects
- Geometry and connectivity
- fast to render because of optimized GPUs

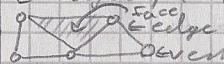
Polygon

- Vertices: $v_0, \dots, v_{n-1}$

- Edges: $\{(v_0,v_1), \dots, (v_{n-2}, v_{n-1})\}$

- planar and non-self-intersecting

- Polygon Mesh: set of connected polygons: $M = \langle V, E, F\rangle$, V: Vertices, E: Edges, F: Faces



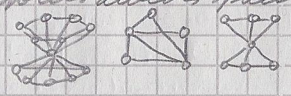
- Every Edge belongs to at least one polygon

- The intersection of two polygons in M is either empty, a vertex or an edge.

- Vertex degree (Valence): number of connected edges

- Boundary: all edges belonging to only one polygon

- Manifold Mesh:
  - Surface locally homeomorphic to a disk
  - closed manifolds divides space into two
  - not allowed:



- Store geometry + topology
  - Geometry: vertex location
  - Topology: how vertices are connected
  - Attributes: Normal, color, etc.

- Mesh data structure has to support:
  - Rendering
  - Geometry queries (e.g. What are the vertices of face F)
  - Modification (e.g. remove vertex v)

Texture Mapping

- Enhance details without adding geometric complexity.

- Issues: • How to map? • Anti-Aliasing and filtering • level of detail

- Texture mapping:
  - One-to-one: Point (u,v) on texture maps to x(u,v), y(u,v), z(u,v) of the model → needs some kind of parameterization • $\begin{bmatrix}\sin(u)\sin(v)\\ \cos(u)\\ \cos(v)\sin(v)\end{bmatrix}$
  - ↳ for sphere: $\begin{bmatrix}u\\ v\end{bmatrix} \mapsto$
  - ↳ desired properties: - low distortion - bijective mapping - efficient to compute

- low-pass filter to avoid aliasing
  - ↳ typical: isotropic 2D Gaussian $f(x,y) = (2\pi\sigma^2)^{-1}\cdot\exp(-\frac{x^2+y^2}{2\sigma^2})$
  - ↑ for screen space
  - ↳ for texture space use: $f(x,y) = (2\pi\sigma_x\sigma_y)^{-1}\cdot\exp(-(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2}))$

Light maps

- simulate a local light source by a alpha mask which is added to the texture

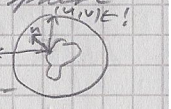
- can be precomputed and dynamically adapted

Environment maps

- used for rendering reflective objects efficiently

- Texture coordinates are computed by intersecting the reflected ray with the surrounding sphere

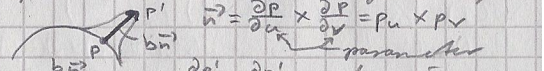
- Using a cube instead of a sphere makes the computation simpler



Bump Mapping

- Perturb ("storm") surface normal according to texture

- represents small-scale geometry



$\vec{n} = \frac{\partial p}{\partial u}\times\frac{\partial p}{\partial v} = p_u \times p_v$

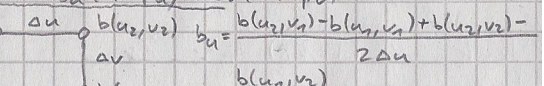
$p' = p + \frac{b\vec{n}}{\|\vec{n}\|}$

$n' = \frac{\partial p'}{\partial u}\times\frac{\partial p'}{\partial v}$

$\frac{\partial p'}{\partial u} = \frac{\partial p}{\partial u} + \frac{\partial}{\partial u}\cdot\frac{(b\vec{n})}{\|\vec{n}\|}$

vectors: $p, p', \vec{n}, n'$

$n' = n + b_u(n\times p_u) + b_v(n\times p_v)$



$b_u = \frac{b(u_2,v_1) - b(u_1,v_1) + b(u_2,v_2) - b(u_1,v_2)}{2\Delta u}$

$b_v = \frac{b(u_1,v_2) - b(u_1,v_1) + b(u_2,v_2) - b(u_2,v_1)}{2\Delta v}$

- limitations: - no bumps on the silhouette, - no self-occlusions, - no self-shadowing

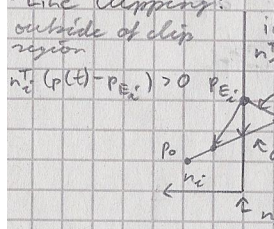
→ displacement mapping:

⇒ do not change the normal, but perturb the geometry directly using the texture → solves the limitations but also more computation needed.

Mip-Mapping
- Store down-sampled versions of texture (multiple versions are stored)
- use low-res for far away objects
- interpolate for in-between depths
- avoids aliasing
- improves efficiency
- compute lower resolution by weighted average: 

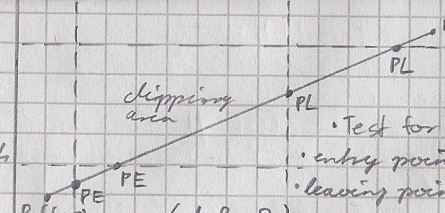
Perspective Interpolation
- from texture coordinates of vertices to texture coordinates of pixels
- linear interpolation in screen-space.
⇒ does not work: linear variation in world coordinates yields non-linear variation in screen coordinates

Clipping
- removing parts of objects outside the viewing frustum
→ saves computation, → memory constancy, → stability: no division by zero
- "Culling": rejects via bounding volumes. bounding volume hierarchies
- "Clipping": partially visible objects need to be clipped
- Line Clipping:



Idea: compute intersection in 1D parameter space of the line
$p(t) = p_0 + (p_1 - p_0)\cdot t \quad t \in [0,1]$
↳ parametric form of the line
- intersection point: $n_i^T(p(t)-p_{E_i})=0$
insert $t = \frac{n_i^T(p_0-p_{E_i})}{-n_i^T d}$
$d = p_1 - p_0$

- classification of entry points ("PE") and leaving points ("PL")

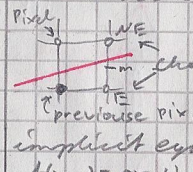


- Test for all $t \in [0,1]$
  - entry point: $n^T d < 0$
  - leaving point: $n^T d > 0$
- compute max/min:
$t_E = \max(t_{E_i}, p_{E_i})$, $t_L = \min(t_{L_i}, p_{L_i})$
→ if $t_E > t_L$ ⇒ no relevant intersection
- Lookup table:

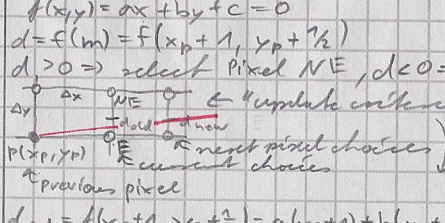
edge i	normal n_i	p_{E_i}	
$x = x_{min}$	$(-1, 0)$	(x_{min}, y)	left
$x = x_{max}$	$(1, 0)$	(x_{max}, y)	right
$y = y_{min}$	$(0, -1)$	(x, y_{min})	bottom
$y = y_{max}$	$(0, 1)$	(x, y_{max})	top

- extension of Algo for polygons:
(i) walk around polygon
(ii) for each edge output endpoints of visible part
(iii) sometimes turning vertex needed
- 3D clipping: problem: frustum is not rectangular
↳ solution: clip after perspective division in homogeneous coordinates

Scan conversion (rasterization)
- generate discrete pixels
- approximate by finite number of pixels
- Bresenham line:
- select closest pixel at each intersection



implicit equation of straight line:
$f(x,y) = ax + by + c = 0$
$d = f(m) = f(x_p + 1, y_p + \frac{1}{2})$
$d > 0 \Rightarrow$ select Pixel NE, $d < 0 \Rightarrow$ pixel E

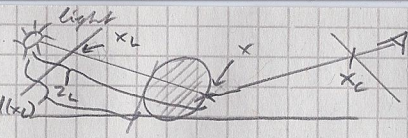


$d_{old} = f(x_p+1, y_p+\frac{1}{2}) = a(x_p+1) + b(y_p+\frac{1}{2}) + c$
$d_{new} = f(x_p+2, y_p+\frac{1}{2}) = a(x_p+2) + b(y_p+\frac{1}{2}) + c$
$d_{new} - d_{old} = a = \Delta y$

$d_{old} = f(x_p+1, y_p+\frac{1}{2})$
$d_{new} = f(x_p+2, y_p+\frac{3}{2})$
$d_{new} - d_{old} = a+b = \Delta y - \Delta x$

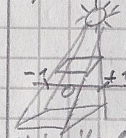
Visibility / Shadows
- simple solution: paint from furthest away to nearest (back to front): "Painter's Algorithm"
↳ Problem: cyclic overlap and intersections
- Solution: Z-Buffering: Store depth to the nearest object for each pixel:
(i) initialize all pixels with ∞
(ii) for each polygon: if z value of a pixel for this polygon is smaller than stored z value, replace the stored value with new, smaller value
↳ problem: limited resolution
↳ resolution is not linear
↳ set near plane far from camera

- Shadows important for:
- depth cue, - scene lighting, - realism
- Basic Shadows:
i) - draw projection of the object as shadow on the ground
↳ limitation: self shadow, shadow on other objects, curved surfaces
ii) - projective texture shadow:
(1) render obstacle (has a shadow) and receiver (of the shadow)
(2) compute b/w image of the obstacle from light
(3) Use image as projective texture
↳ limitations: need to specify obstacle + receiver, no self-shadows
- Shadow maps (high-end production and games):
- compute depth from camera
- compute depth from light
↓ for each pixel in the camera plane:
(1) compute the point in world coordinates (2) project point onto light plane (3) compare $d(x_L)$ (shadow map) and z_L
(4) if $d(x_L) < z_L$, x is in shadow



- limitations: for visible point $d(x_L) < z_L$ would lead to self-shadowing
→ solution: add bias: $d(x_L) + bias < z_L$
- Limitation: a point to shadow can be outside the field of view of shadow map → solution: use cubical shadow maps or spot lights
- limitation: choosing good bias can be tricky
- limitation: undersampling the shadow map → aliasing
↳ solution: depth filtering is bad idea. instead filter the result of our test $d(x_L) + bias < z_L$ by taking weighted average of comparisons

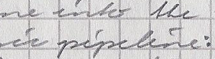
- Shadow volumes:
- explicitly represent the volume in shadow → if polygon is inside, (⇒ polygon is in shadow
- Algo: (1) shoot a ray from the eye (2) increase/decrease counter each time volume boundary is intersected
- if counter > 0, primitive is in shadow, otherwise (=0) primitive not in shadow



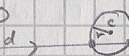
- optimization: use silhouette edges only
- limitation: introduces a lot of new geometry, - expensive to rasterize long skinny triangles, - objects must be watertight to use the silhouette optimization, - rasterization of polygons sharing an edge must not overlap and not have gap

Ray Tracing

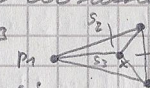
- Forward Ray Tracing: start from light source, until going through image plane / camera
- Backward Ray Tracing: start from eye, going through image plane into the scene
- Basic pipeline: Generate ray → Intersection → Shading
- Extensions: - Anti-Aliasing: multiple rays per pixel (supersampling)
- Ray equation: $r(t) = o + td$



- Sphere: Equation: $\|x - c\|^2 - r^2 = 0$
  (point of interest x, center c, radius r)
  ↳ insert ray equation ⇒ solve for t:
  $\|o + td - c\|^2 - r^2 = 0$



- Triangle: Barycentric coordinates:
  $x = s_1 p_1 + s_2 p_2 + s_3 p_3$
  ↳ intersection of ray with triangles plane:
  $(o + td - p_1) \cdot n = 0$, $t = (o - p_1) \cdot n / d \cdot n$
  where $n = (p_2 - p_1) \times (p_3 - p_1)$
  → compute s_1, s_2, s_3
  → test $s_1 + s_2 + s_3 = 1$ $(0 \le s_i \le 1)$



- Shading: physically exact shading too costly. Simplify assumptions:
  - Surface reflectance: diffuse, specular, ambient, transparency terms
  - Shadows: shadow rays to determine if a hit point is in shadow or not
- Performance: complex scenes done with primitive algorithm costs $O(\#\text{rays} \times \#\text{objects})$
  → solution: fewer intersections:
  - uniform grids
  - space partitioning trees (data structure)

Uniform grids:

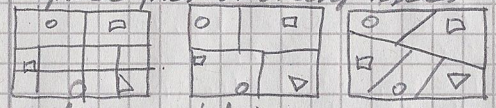
(1) Preprocessing:
- compute bounding box
- set grid resolution
- rasterize objects
- store references to original objects

(2) traversal:
- incrementally rasterize ray
- stop when intersection happens

advantages:
+ fast to build
+ easy to code

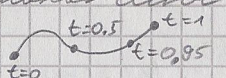
disadvantages: non-adaptive to scene geometry

- Space partitioning trees



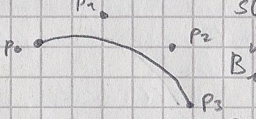
geometry representations

- Planar curve



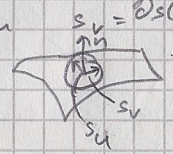
$f: X \to Y$, $X \subseteq \mathbb{R}^m$, $Y \subseteq \mathbb{R}^n$
$m = 1, n = 2$
$s(t) = (x(t), y(t))$

- Circle: $p: \mathbb{R} \to \mathbb{R}^2$
  $p(t) = (x(t), y(t)) = r \cdot (\cos(t), \sin(t))$
  $r \in [0, 2\pi]$
- Bezier curves



$s(t) = \sum_{i=0}^{n} p_i B_i^n(t)$
$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$

- Space curves in 3D:
  $f: X \to Y$, $X \subseteq \mathbb{R}$, $Y \subseteq \mathbb{R}^3$
  $s(t) = (x(t), y(t), z(t))$
- Surfaces: $f: X \to Y$, $X \subseteq \mathbb{R}^2$, $Y \subseteq \mathbb{R}^3$
  $s(u,v) = (x(u,v), y(u,v), z(u,v))$
- Sphere:
  $s(u,v) = r \cdot (\cos(u)\cos(v), \sin(u)\cos(v), \sin(v))$, $(u,v) \in [0, 2\pi) \times [-\pi/2, \pi/2]$
- Bezier surfaces:
  $s(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} p_{i,j} B_i^m(u) B_j^n(v)$
- normal and tangent plane
  $s_u = \partial s(u,v)/\partial u$, $s_v = \partial s(u,v)/\partial v$
  $n = \frac{s_u \times s_v}{\|s_u \times s_v\|}$



Parametric curves + surfaces:
+ easy to generate points on curve/surface
+ easy point-wise differential properties
+ easy to control by hand
- hard to determine inside/outside
- hard to determine if a point is on curve/surface
- Hard to generate by scene engineering

- Implicit curves + surfaces:
  - Planar curve: $S = \{x \in \mathbb{R}^2 \mid f(x) = 0\}$
  - Surface in 3D: $S = \{x \in \mathbb{R}^3 \mid f(x) = 0\}$
  - outside $\{x \mid f(x) > 0\}$
    inside $\{x \mid f(x) < 0\}$
  - Circle: $f(x,y) = x^2 + y^2 - r^2$
  - Sphere: $f(x,y,z) = x^2 + y^2 + z^2 - r^2$
  - surface normal:
    $\nabla f(x,y,z) = (\partial F/\partial x, \partial F/\partial y, \partial F/\partial z)^T$
  - circle normal: $\nabla f(x,y,z) = (2x, 2y, 2z)^T$

↳ + easy to determine inside/outside
+ easy to determine if point is on curve / surface
+ easy to combine
- hard to generate points on curve / surface
- limited set of surfaces
- does not lend itself to (real-to-c) rendering

- Point Set Surfaces:
  + robust to noise
  + easy to determine inside/outside
  + easy to determine if point is on curve / surface
  + easy to generate point on curve / surface
  + suitable for reconstruction from general data
  + direct real time rendering
  - not efficient to use in some modeling tasks

Aliasing: sample frequency < target frequency / 2
- stationary wheel: sampling frequency = = rotational speed modulo features of wheel
- wheel moving wrong direction: sampling frequency slightly smaller than rotational speed modulo features of wheel

- Optical flow: Lucas-Kanade algo: assumptions:
  (i) Brightness constancy: $I_x u + I_y v + I_t = 0$
  (ii) Single velocity (u, v) for all pixels in patch
  (iii) Movement is slow

- Phong Reflection based on
  (i) Ambient light
  (ii) Diffuse
  (iii) Specular
  → Specular based on camera
  ↳ no camera → no specular

- orthogonal
  → $\vec{a} \cdot \vec{b} = \begin{bmatrix} a \end{bmatrix} \begin{bmatrix} b \end{bmatrix} = 0$
  → $u \times v = \vec{0} = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 u_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix}$

- Quaternion multiplication table:

·	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

- Perspective division after Projection with A:

$c' = A \cdot c$
↑ new coords. after projection

$c'' = \begin{bmatrix} c'_1/-c_3 \\ c'_2/-c_3 \\ c'_3/-c_3 \\ 1 \end{bmatrix}$ ← perspective division done
↑ original c
after projection

- I: image, h: filter, symmetric
  $\Rightarrow I \circ h = I * h$
  $h' = 2 \cdot h \Rightarrow I * h' = 2(A \circ h)$