Visual Computing HS19

Image Segmentation

Thresholding: $B(x,y) = \begin{cases} 1 & \text{if } I(x,y) < T \\ 0 & \text{otherwise} \end{cases}$

for binary img. Trial & Error to get T.

Chromakeying Filter for BG color:

$I_\alpha = |I - g| > T$; g = RGB color

ROC = Performance of binary classifyers

$TP + FN = \#P$; $TN + FP = \#F$

sensitivity $= \frac{TP}{P}$ specificy $= \frac{FP}{N}$

Curve $= \frac{TP}{P}$  $\frac{FP}{N}$

Pixel connectivity ⌗ / ⌗ 4/8 Neighb.

Connected Component Labeling

- Scan Row by Row → Look at Neighbours → if not labeled := new label else copy labels
- Build equiv. list of multilabeled pixels, do 2nd run and „merge" equivalent lbls

Region growing Start from a seed point, add pixels that satisfy criteria, repeat until no more pixels are added.

Background subtraction

$I_\alpha = |I - I_{bg}| > T$ or better:

$I_\alpha = \sqrt{(I - I_{bg})\, \Sigma^{-1} (I - I_{bg})} > T$, where

Σ = BG Pixel appr. cov. Matrix (computed individually for each pixel, e.g. from Gaussian Mixture Model)

Image as binary set

$I = \{(x_1, y_1), \dots\}$ S = Structuring element

S fits I at x: All of S are in I

S hits I at x: Some of S are in I

S misses I at x: None of S are in I

Erosion $E = I \ominus S$

$E(x) = 1$ if S fits I at x; 0 else

Dilation $D = I \oplus S$

$D(x) = 1$ if S hits I at x; 0 else

Opening $I \circ S = (I \ominus S) \oplus S$

Closing $I \bullet S = (I \oplus S) \ominus S$

Thinning $I \oslash S = I \setminus (I \otimes S)$

Thickening $I \odot S = I \cup (I \otimes S)$

Exact Match $I \otimes S$; simple templ. matching, 1 at „center" where S fits

Medial Axis transform

Builds a „skeleton", start a grassfire at boundary, skeleton is set of points where two fire fronts meet.

Use $B = \begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$ with $\ominus_n$ = n Erosions

$S_n(X) = (X \ominus_n B) \setminus [(X \ominus_n B) \circ B]$

$S(X) = \bigcup_{n=1}^{\infty} S_n(X)$

Markov random fields

- Field of sites, each with lbl. Label a site with respect to neighboring sites.
- Smoothness: prefer sol. where neighboring sites have same lbl (Regularizer)

$E(y; \theta, \text{data}) = \sum \psi_1(y_i; \theta, \text{data})$ ← data term

$+ \sum_{\text{edges}} \psi_2(y_i, y_j; \theta, \text{data})$ ← smoothness term

Filtering

Linear Map $F[\alpha I_1 + \beta I_2] = \alpha \cdot F(I_1) + \beta \cdot F(I_2)$

Correlation Multiply componentw., sum up:

$I' = K \circ I = \sum_{(i,j) \in N(x,y)} K(i,j) \cdot I(x+i, y+j)$

(Multiply each el of Kernel with el of img underneath, add up)

Convolution punktspiegelung in Mitte, mult. componentwise, add up.

$I' = K * I = \sum_{(i,j) \in N(x,y)} K(i,j) \cdot I(x-i, y-j)$

→ Same as correlation, with revers. Kernel

→ If $K(i,j) = K(-i,-j)$ → $K \circ I = K * I$

Kernel Koord.:

-1,-1	0,-1	1,-1
-1,0	0,0	1,0
-1,1	0,1	1,1

⇒ 3x3: $9N^2$ runtime

Avoid aliasing on Edges

- clip filter → add 0
- reflect image
- wrap around
- vary filter near edge
- copy edge

Some Kernels

box blur: $\frac{1}{9} \cdot \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$

sharpen: $\begin{matrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{matrix} - \frac{1}{9} \cdot \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$

prewitt: $\begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix}$ sepr. $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}^T$

sobel: $\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$ separable $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}^T$

laplacian: $\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$

High pass: $\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$

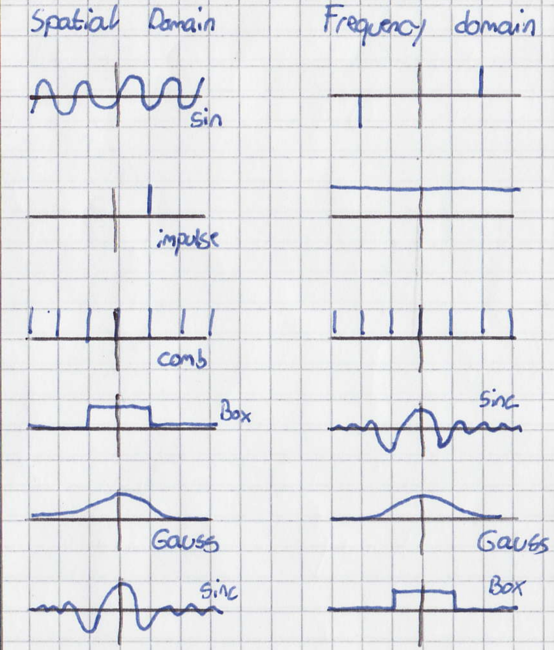
Separable Kernel 3x3: $6N^2$ runtime

If $K(m,n) = f(m) \cdot g(n)$

Gaussian used for smoothing, depends on σ and window size. Is symmetric, but not seperable

Poisson $p(k) = \frac{\lambda^k \cdot e^{-\lambda}}{k!}$

Fourier Transformation



- Linear; inverse exists
- scale func. down → scale trans. up

Convolution Theorem

- FT(convol.) = Prod(FTs):

  $\mathcal{U}(f ** g) = F \cdot G$

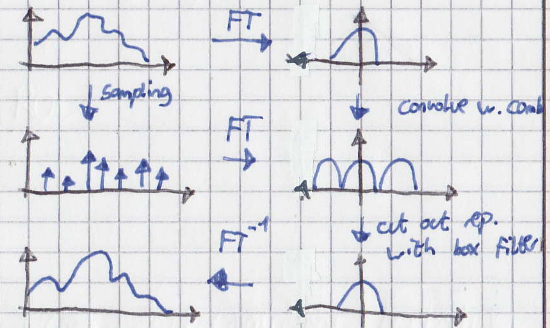
- Conv(FTs) = FT(prod):

  $F ** G = \mathcal{U}(f \cdot g)$

Aliasing & Sampling: Suppress high frequencies before sampling (e.g. convolve with gaussian)

Nyquist sampling theorem

Sampling freq $\geq 2x$ highest freq.



Motion Blur Transform each „light dot" into a line along the x_1 - Axis

Edge / Corner detection

High Gradient = Edge

Laplacian Operator detect zero-crossing in second derivative ← invariant to rotation

Local Gradient:

$$|grad(x,y)| = \sqrt{\left(\frac{df}{dx}\right)^2 + \left(\frac{df}{dy}\right)^2}$$

Laplacian of Gaussian $LoG(x,y)$: Blur with Gaussian, edge via Laplace

Laplacian in discrete: Approx. with $\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$ or $\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$

- Sensitive to high freq. / noise,
  ↳ blur first (LoG)

Canny Edge detector

1) Smooth image (Gaussian)
2) Compute gradient magn. & direction (Sobel, Prewitt, ...)

$$M(x,y) = \sqrt{\left(\frac{df}{dx}\right)^2 + \left(\frac{df}{dy}\right)^2} \quad \alpha(x,y) = \tan^{-1}\left(\frac{df}{dx} \Big/ \frac{df}{dy}\right)$$

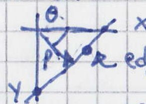
3) Apply nonmax. suppr. to grad. magn. img
4) Double thresholding to detect strong & weak edge pixels
5) Reject weak edges not connect. to strong.

Nonmaxima Suppression

1) Categorize gradient to / \ | —
2) If magnitude is smaller than neighboring ones in category (direction), don't take it

Hough transform

Parameterize points in the image by $\rho = x\cos\theta + y\sin\theta$



Map that parameter space onto a ρ-θ diagram → results in a sinusoid in the (θ, ρ) plane. Points on the same line in x,y intersect in (θ, ρ), giving the polar params of the line. If local maxima in polar plane do not change, noise is no issue.

Local Displacement Sensitivity

$$S(\Delta x, \Delta y) = (\Delta x, \Delta y)\, M \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

$$SSD = \underbrace{\quad}_{\Delta^T} \; M \; \underbrace{\quad}_{\Delta}$$

with

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

$f_k(x,y)$ = Gradient in direction k

$SSD = \Delta^T M \Delta$, find $\min \Delta^T M \Delta$; $\|\Delta\| = 1$, i.e. Max. Eigenvalues of M

Keypoint / Harris Edge detector

M same as above, measure „cornerness":

$$C(x,y) = \det(M) - k \cdot (\text{trace}(M))^2 = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2).$$

If $\lambda_1 \gg \lambda_2$ or $\lambda_2 \gg \lambda_1$ → Edge

Both large → corner

Both small → flat region

→ Invariant to intensitivity shift

→ Invariant to shift & rotation

→ Invariant to brightness offset

→ Not invariant to scaling

Better localization of corners by weighting with Gaussian → give more importance to center pixels

Scale inv. Feature transf. SIFT

Recover features from images with posit., orientation & scaling. Used for image recognition. DoG = Difference of Gaussian

- Position: strong response of DoG
- Scale: DoG over scale space
- Orientation: Compute gradient for each pixel, create histogram. Peak (= most pixel gradients in that direct.) = canonical direction.

Sift descriptor: Create array of orientation histograms of image parts. 4x4 is the final hist. size, with 8 directions → 128 dimensions.

Unitary Transforms

$$A^{-1} = A^{*T} = A^H$$

Linear operator $O[\alpha_1 \cdot \vec{f}_1 + \alpha_2 \cdot \vec{f}_2] = \alpha_1 \cdot O[\vec{f}_1] + \alpha_2 \cdot O[\vec{f}_2]$

Energy conservation: for unit. trans. $\vec{c} = A\vec{f}$ (with $A^H = A^{-1}$):

$$\|\vec{c}\|^2 = c^H c = f^H A^H A f = f A^{-1} A f = f^H f = \|f\|^2$$

→ Every unitary transform is simply a rotation of coord. system; Vector lenghts („energies") are preserved

Autocorrelation Matrix

Img collection $F = [f_1\ f_2 \dots f_n]$ (n vectors, where each is an img.)

$R_{ff} = E[f_i \cdot f_i^H] = \frac{F \cdot F^H}{n}$ image collection auto-cor. function

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

Eigenmatrix of Autocorrelation

Φ of R_{ff}: unitary, cols = Eigenvect. of R_{ff}

$R_{ff}\Phi = \Phi\Lambda$ ← Eigenval. Matrix

Karhunen-Loeve / PCA

Unit. Transform with $A = \Phi^H$, with cols. of Φ sorted by decreasing Eigenval.

$R_{cc} = A R_{ff} A^H = \Phi^H R_{ff} \Phi = \Phi^H \Phi \Lambda = \Lambda$

→ Energy concentration, most energy (vector length) in first j components, with arbitrary j.

→ Mean squared approx. error by choosing only first j components is minimized

PCA Algorithm

- Center data
- Normalize data

} – preprocessing

- Compute covar. matrix Σ of data
- Perform Eigen decompos. of Σ
  ↳ EV give directions that describe variation of data. Corresponding EW describe magnitude of variation.
- Covar. Matrix Σ is symmetric
  ↳ components are orthogonal, EV are real-valued

Compressing via PCA

K = dimension of compr. image

$I \in \mathbb{R}^n$ = Image ; $\bar{I} \in \mathbb{R}^n$ = mean img

n = Size of image

$I_K \in \mathbb{R}^K$ = Compressed img (K factors for the K Eigenimages)

$\Phi \in \mathbb{R}^{n \times K}$ = trunc. Eigenmatrix of covariance (Eigenimages).

- Compression formula: $I_K = (I - \bar{I})\,\Phi$
- Decompression: $\hat{I} = I_K \Phi^T + \bar{I}$

Storage Overhead:

- dataset mean $\bar{I}$ = n (img. size)
- trunc. Eigenmatrix Φ = $n \times K$
- Compressed images $b \cdot K$, where b = number of images to store

Eigenfaces

- PCA on training images (each img. is convert. to a row vector; resulting matrix is centered and normalized [subtracting avg. img.]):

$$\bar{I} = \frac{1}{n} \sum_{i=1}^{n} I_n \; ; \; \hat{I}_i = I_i - \bar{I}$$

$$[\hat{I}_1\; \hat{I}_2 \ldots] = U \Sigma V^T = \boxed{U}\,\underset{\Sigma}{\boxed{\diagdown}}\,\underset{V^T}{\boxed{}}$$

] via SVD

- To compress, only take the k EV with largest EW → $\boxed{U_k}\,\underset{\Sigma_k}{\boxed{\diagdown}}\,\boxed{}\,V_k^T$
- $c_i = \Sigma_k (V_k^T)_i$ „Coefficient" of face i

→ Can store K coeff. for an img.

→ Restoration: mean_face $+ \sum_{i=1}^{k} c_i \cdot$ eigenface$_i$, where EF_i = i-th col of U_k

- Eigenfaces are sensitive to lighting variation!

Pyramids & Wavelets

Scale-Space Representation

From an original signal $f(x)$, generate a family of signals $f^t(x)$, where succesively fine-scale info is suppressed (i.e. smoothing with Gauss)

Pyramid: Scale down img at each level

↳ Search for correspondence

↳ Edge tracking

↳ Control of detail & computation

Gaussian pyramid

- Smooth at each lvl with gaussian, because Gauss $*$ Gauss = Gauss
- Gauss = Low Pass filter → representation is redundant

Laplacian Pyramid

- Save difference between upsampled gauss lvl and gauss pyramid lvl
- Band-pass filter → each lvl represents spatial freq. unrepresented at other lvl
- Analysis: reconstr. gauss pyramid, take top layer

Oriented Pyramids

Laplacian Pyramids are orient. independent.

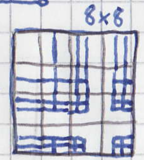
→ Apply an oriented filter (kernel) to determine orientations at each layer

→ this represents img. info at a particular scale & orientation

Wavelet transform: Decomposing a signal by applying a two-band filter to lowpass band of prev. stage into multiple signals. Result back in old signal when added together.

Haar-Transform

- Real & orthogonal
- 2D basis img of transf.:



Optical Flow

Basic Assumptions

Brightness constancy; small motion; spatial coherence

Brightness Const. Assumption

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

„Brightness doesn't change"

- $\frac{dI}{dt} = \frac{\delta I}{\delta x}\frac{dx}{dt} + \frac{\delta I}{\delta y}\frac{dy}{dt} + \frac{\delta I}{\delta t} = 0$

„Change of intens. is comp. by shift in space"

⇒ Opt. Flow: $u = \frac{dI}{dx}$; $v = \frac{dI}{dy}$

⇒ Loc. Gradi.: $I_x = \frac{dI}{dx}$; $I_y = \frac{dI}{dy}$; $I_t = \frac{dI}{dt}$

⇒ $I_x u + I_y v + I_t = 0$

↳ 2 Unknowns, 1 Equation

Aperture Problem: Can not determine where a point on a line moves (e.g. Barberpole), because we have 1 Equation in 2 unknowns.

Horn & Schunck Algorithm

- Add a smoothness „regularizer":

$$e_s = \iint \left((u_x^2+u_y^2) + (v_x^2+v_y^2)\right)dxdy$$

- besides OF constraint:

$$e_c = \iint (I_x u + I_y v + I_t)^2 dxdy$$

⇒ minimize $e_s + \lambda e_c$

→ Errors at boundaries; Example of Regulariz.

Lucas-Kanade

- Assume same displac. for a patch

$$\underbrace{\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ \vdots & \vdots \\ I_x(p_n) & I_y(p_n) \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{x} = \underbrace{-\begin{bmatrix} I_t(p_1) \\ \vdots \\ I_t(p_n) \end{bmatrix}}_{b} \quad *$$

Want to min. $(Ax-b)^2$ → Least Squares with $A^TAx = A^Tb$:

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

Derivation of Lucas-Kanade

1) Bright. const.: $I(x,y,t) = I(x+u, y+v, t+1)$
2) Taylor. exp. with small motion:

$I(x+u, y+v, t+1) = I(x,y,t) + I_x u + I_y v + I_t$

Bc. of 1), we get $I_x u + I_y v = -I_t$

⇒ Single eq. 2 unkn. Use spacial coh. to get equation * from above.

- A^TA is invertible

+ Good for textured areas

− Not good for large (>1Px) Motion; if point doesn't move like neighbors; if brightness const. doesn't hold

Iterative Refinement

- Coarse-to-fine by using image pyramids (to capture large motion)
- Estim. pixel velocities, warp one img. towards other using estim. flow, repeat

Video Compression

DCT block based (discrete) variant of DFT → + real numb./fast implem.

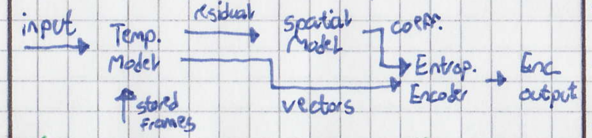
Interlaced Video: use 2 temp. shifted half-img. to increase freq. to 50 Hz

Temporal Redundancy: pred. current img. based on prev. − doesn't work when scene changes often/high motion

Intra-coded frame: independent of others

Predictively-coded: based on prev. frame

Bi-directionally: based on prev & next



Motion compensated prediction

Idea: Video into moving obj. → descr. motion

Practice: Block Matching motion est.

- All pixels inside a block have same motion

1) Divide curr. frame into non-overlapping $N_1 \times N_2$ - Blocks
2) For each block: find best matching block in reference frame

⇒ Use best matching blocks of ref. frame as prediction of blocks in curr. frame

Motion Vector: Expresses rel. hor & vert offset (mv_1, mv_2) of a block

Motion Vector Field: Collection of motion vectors of all blocks in a frame

Fast Motion Est. Search: Perform coarse-to-fine search; Next step is centered at best match of prior step

Half-Pixel MV: Use fractional MV to represent sub-pixel motion

+ Can capture half-pixel motion; Averaging effect; Reduces noise → improved compression

Block Matching Advantags/Disadv.

+ Good, robust perform. for compression
+ Resulting MV field easy to represent
+ Simple, periodic structure

− Assumes translat. mot. model → Breaks down for complex motion

− Often prod. block artifacts (OK for DCT)

Sparsity & Texture

Dictionary: Normalized set of basis vect. D is adapted to img x if x can be represented by a few vect. of D ($D \cdot a \approx x$, with a being the sparse code

⇒ Sparse represent. can be used to represent data, but not noise → use for denoising by introducing a sparsity term (analog ML)

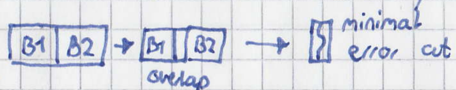
Represent Texture: Find meaningful subelem. with meaningfull repetition → Use oriented filters

Texture as Pyramid: form an oriented pyramid (e.g. Laplacian Pyramid), apply a number of oriented filters → represents img info at particular scale & orientation

Chaos Mosaic: tile image; pick random blocks and place in random location; smooth edges. Works well for random, unstructured textures

Efros-Leung: $P(p \mid N(p))$: search src img for similar neighb. of pixel when generating a new pixel → consider spatial constr.

Image Quilting: $P(B \mid N(B))$: consider a block as unit. Faster: Block by block:



Computer Graphics

Drawing Triangles

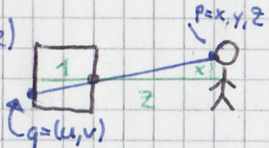
Coverage $C(x,y) = 1$ if triangle contains (x,y); 0 else

Supersampling: Take multiple samples per pixel, average samples to get color

Point-in-Triangle: For each edge, test if point is inside: $=0 \to$ Edge; $<0 \to$ inside

$E_i(x,y) = (x - X_i)\,dY_i - (y - Y_i)\cdot dX_i$, with $dX_i = X_{i+1} - X_i$ (Edge)

Camera Coord.



$v = y/z \quad u = x/z$

Transforms

Linear Map takes lines to lines; keeps origin

· $f(u+v) = f(u) + f(v)$; $f(\alpha v) = \alpha \cdot f(v)$

Affine Map: $f(x) = ax + b$ (does not go trough origin → not linear map)

Basis change: u has coord. in B with basis vectors $\hat{\imath}$ and $\hat{\jmath}$. It is in A:

$\underbrace{f(u_B)}_{A} = f(u_1 \hat{\imath} + u_2 \hat{\jmath}) = u_1 \cdot f(\hat{\imath}) + u_2 \cdot f(\hat{\jmath})$

⇒ Only need to find out how to represent $\hat{\imath}$ and $\hat{\jmath}$ in A!

Scale (Linear): Uniform: $S_\alpha(x) = \alpha \cdot x$

Non-Uniform: $S(x) = x_1 a e_1 + x_2 b e_2$

Rotation about Origin (Linear)

$R_\theta(x) = x_1(\cos\theta, \sin\theta) + x_2(-\sin\theta, \cos\theta)$

Shear: $\square \to$ ▱ (a) $\quad H_a(x) = x_1\begin{bmatrix}1\\0\end{bmatrix} + x_2\begin{bmatrix}a\\1\end{bmatrix}$

Homogenous Coord: „Lift" to higher dim:

Hom. Point: $3D = [x,y,z] \to 3D\text{-}H = [x,y,z,1]$

Hom. Vector: $3D = [x,y,z] \to 3D\text{-}H = [x,y,z,0]$

Translation (affine): $T(x) = x + b$

→ in Homog. Coord., translation is linear

Composing Linear transforms

→ done via Matrix Multiplication

As 2D Matrices

$S(x) = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \cdot x \qquad R_\theta(x) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot x$

$H_a(x) = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \cdot x \qquad T(x) \overset{2DH}{=} \begin{bmatrix} x_1 + b_1 \\ x_2 + b_2 \\ 1 \end{bmatrix}$

3D Transforms $c\theta = \cos\theta$; $s\theta = \sin\theta$

$\begin{bmatrix} 1 & 0 & 0 \\ 0 & c\theta & -s\theta \\ 0 & s\theta & c\theta \end{bmatrix}$ Rot. about x $\quad \begin{bmatrix} c\theta & 0 & s\theta \\ 0 & 1 & 0 \\ -s\theta & 0 & c\theta \end{bmatrix}$ Rot. about y $\quad \begin{bmatrix} c\theta & -s\theta & 0 \\ s\theta & c\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$ Rot. about z

$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$ Translation 3DH $\quad \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}$ Scaling $\quad \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ Shear in x

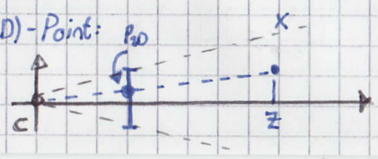
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$ Reflection x $\quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$ Reflection y

Perspective Projection

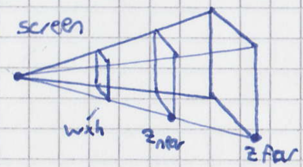
Basic PP: Input: 3D Point $p = (x,y,z)$

Output: PP(2D)-Point: p_{2D}

$p_{2D} = (x/z, y/z)$



View Frustum: Region in space that will appear on screen



θ = Field of view

$h = 2 \cdot \tan(\frac{\theta}{2})$

r = aspect ratio = width / height; $f = \cot(\frac{\theta}{2})$

$$\begin{bmatrix} f/r & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_f + z_n}{z_n - z_f} & \frac{2 \cdot z_f \cdot z_n}{z_n - z_f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Geometry

Implicit Representation

- Shape of object is given by a relation

+ Inside/outside check is easy

- Sampling is hard

Algebraic Surface: Geometry given by polynomial where $f(x,y,\ldots) = 0$

Constructive Solid Geometry

Build complicated shapes via Bool oper.

Distance function: Gives the dist. to closest point on object

Level set: Store grid of values approximating the distance function, interpolate to find surface

Fractals & L-Systems: described by special „language" (similar to derivation trees in other applications)

Explicit Representation

- Geometry is given directly

+ Sampling is usually easy

- In/Out is difficult

Point Cloud: Store single points

Polygonal Mesh: Store vertices & polygons (e.g. triangles)

Triangle Mesh: Store vertices as triplets of coordinates, triangles as triplets of indices (of vertices)

Energy on area A: $E = \frac{\Phi}{A'} = \frac{\Phi \cdot \cos\alpha}{A}$

Φ = Flux; A' = Area A with angle α

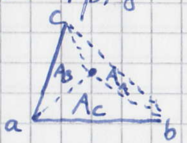
N-dot-L shading: $\max(0, \text{dot}(N, L))$

N = surface normal; L = unit dir. to light

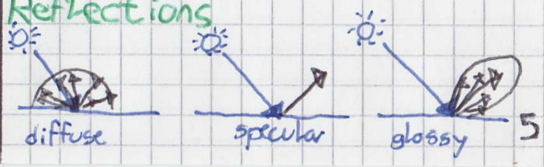
Phong-Shading: Similar to N-dot-L, → see page 8 but instead of using one normale for the whole polygon, it uses the normals of the vertices and interpolates them.

Interpolating Attributes

$f_x = \alpha a + \beta b + \gamma c$; with α, β, γ obtained by $\alpha = A_A/A$; $\beta = A_B/A$; $\gamma = A_C/A$ and $\alpha + \beta + \gamma = 1$



Reflections



Texture

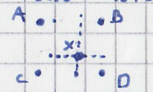
Aliasing due to undersampling ⇒ use pre-filtered textures

Magnification Screen area ⇒ small texture area. Interpolation required

Minification Screen area ⇒ large texture area. Averaging required

Mipmap: Store pre-filtered texture in some „region". The level d of the pyramid can be computed using diff. between screen pixels.

Bi-Linear: d is „clamped" to next lvl. Interpolates value with 4 neighboring values:



+ Better Texture
- Reads 4 texels

Tri-Linear: d is continuous, it interpolates between different mipmap lvls. Even smoother, but more comput. cost

↳ Both **Isotropic**, will „overblur" to avoid aliasing. Use **anisotropic filtering** to avoid this → higher comput. cost

Texture sampling operation

1) Compute u, v from screen sample x, y
2) Compute $\frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}$ from screen-adjacent samples
3) Compute d from differentials
4) Convert norm. texture coord. (u, v) to texture coord $texel_u$, $texel_v$
5) Compute required texels in window of filter
6) Load required texels (e.g. 8 for tri-linear)
7) Perform interpolation (e.g. Trilinear

Graphic Pipeline

Z-Buffer stores depth information of the objects in the scene

Opacity is represented as the α-Channel

Opaque Images: Composite img B with opacity α_B over image A with α_A:

$$C = \alpha_B B + (1 - \alpha_B)\,\alpha_A A$$

$$\alpha_C = \alpha_B + (1 - \alpha_B)\,\alpha_A$$

Graphics Pipeline: Vertex Processing → Primitive Processing → Rasterization (Fragm. Gen.) → Frag. Proc. (Shading) → Screen sample operations

Mix of opaq. & transp. triang.

1) Render opaque surfaces as normal
2) Disable depth-buffer update. Render transp. surfaces back-to-front. If depth-test is passed → triangle is rendered over content of depth-buffer.

Ray-Tracing

Rasterization vs Ray-Casting

Rasterization	Ray-Casting
- Proceeds in triangle order	- Proceeds in screen sample order
- Based on 2D primit.	- Never have to store depth buffer
- Store depth buffer	- Store whole scene

Shadow Mapping (Shadows for Rast.)

1) Render scene from direction of light with depth buffer only
   → Everything „seen" is directly lit
2) Render scene from direction of camera
   → Transform every screen sample to light coordinate frame and perform a depth test (fail = in shadow)

Shadow Ray Tracing

Shoot „shadow" rays towards light source from points where camera rays intersect scene → If unoccluded, point is directly lit by light source

Intersections

- Point-Point: Intersect if they're the same
- Point-Line: Plug point coordinates into line equation $N^T x = c$ (N = unit normal)
- Line-Line: Two lines $ax = b$ & $cx = d$
  ⇒ solve $\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$, where (x_1, x_2) is the point of intersection

Ray-Parametric Equation:

$r(t) = o + td$; $r(t) :=$ point along ray; o = origin; d = unit direction

Ray intersection with impl. surface

- Surface: all points x where $f(x) = 0$

→ replace x with r, solve for t

→ E.g. unit sphere $f(x) = |x|^2 - 1$ → solve $f(r(t)) = |o + td|^2 - 1$ → $t = -o \cdot d \pm \sqrt{(o \cdot d)^2 - |o|^2 + 1}$

Ray-Plane intersection

Plane $N^T x = c$

↳ replace x with ray equation:

$$N^T(o + td) = c \Rightarrow r(t) = o + \frac{c - N^T o}{N^T d} \cdot d$$

Ray-Triangle intersection

- Triangle given by vertices p_0, p_1, p_2

↳ Use barycentric coordinates, plug parametric ray equation into equation for pts on triangle:

$$p_0 + u(p_1 - p_0) + v(p_2 - p_0) = o + td$$

$$\Rightarrow \begin{bmatrix} p_1 - p_0 & p_2 - p_0 & -d \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = o - p_0$$

(solve for u, v, t)

Optimize Raytracing

Add a simple bounding box around each object (simpler to calculate intersection)

→ if ray misses box, it also misses object

Bounding volume hierachy (BVH)

→ Large bounding volumes contain smaller ones
→ if hit, check smaller ones
→ Simple splitting: Uniform grid, choose number of voxels ~ number of primitives
→ Intersection cost $= O(\sqrt[3]{n})$
→ Tree nodes contain regions

K-D Tree

Recursively split space via axis-aligned planes → Interior nodes = splits; Leafs = Regions

Quad- / Octtree

Like unif. grid, but nodes have 4 (quadt.; partitions 2D space) or 8 children (octt.; partitions 3D space).

Keyframing Idea: Specify important events only; computer/assistant fills in the rest inbetween via interpol./approximat.

→ Events can be position, color, light, camera, ...

Interpolation

Piecewise Linear „connect dots"

→ simple, but rough motion „infinite accelerat."

Splines: Mostly use polynomials of third degree for interpolation (smooth; higher order polynomials lead to oscillations at endp.)

Natural Splines: Piecewise made of cubic polynomials p_i. How to determine:

- Interpolation at Endpoint of piece:

$p_i(t_i) = f_i \qquad p_i(t_{i+1}) = f_{i+1}$

- Tangents to agree at Endpoints (C^1 cont.):

$p_i'(t_{i+1}) = p_{i+1}'(t_{i+1}) \qquad i = 0 \dots n-2$

- Curvature to agree at Endpoints (C^2 cont.)

$p_i''(t_{i+1}) = p_{i+1}''(t_{i+1}) \qquad i = 0 \dots n-2$

- Pin down remaining 2 DoF by setting curvature to zero at endpoints

Hermite / Bezier Splines

- Each cubic „piece" is specified by endpoints and tangents at datapoints
- Can get tangent (C^1) cont. by setting tangents to same value on both sides of point (not nessecary! E.g. for sharp corners in vector graphics)
- Endpoints interpolate data:

$p_i(t_i) = f_i \qquad p_i(t_{i+1}) = f_{i+1}$

- Tangents interpolate given tangents:

$p_i'(t_i) = u_i \qquad p_{i+1}'(t_{i+1}) = u_{i+1}$

→ Diff to natural splines: Tangents are given and need to match given

Catmull-Rom Splines

- Specialization of Hermite splines, determined by values (points) alone
- Use difference of neighbors to define tangent:

$$u_i = \frac{f_{i+1} - f_{i-1}}{t_{i+1} - t_{i-1}}$$

B-Splines: Get better continuity & local control, but sacrifice interpolation

- Def. recurs.: $B_{i,1} = \begin{cases} 1 & \text{if } t_i \le t < t_{i+1} \\ 0 & \text{else} \end{cases}$

$B_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} \cdot B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} \cdot B_{i+1,k-1}(t)$

→ Spline is lin. comb. of bases:

$$f(t) = \Sigma_i\, a_i \cdot B_{i,d}$$

Rigging & Animation

Blend Shapes: Simple rig; uses a set of meshes M_i with vertices x_i and blending weights $\alpha = (\alpha_1, \dots, \alpha_n)$. The output is a „blended" mesh, by lin. comb.:

$M = \Sigma_i\, \alpha_i M_i$, i.e. $x^j = \Sigma_i\, \alpha_i x_i^j$

→ Keyframes are blending weights $\alpha(t_i)$, spline used to interpolate weights over time

Cage-based Deformers

Embed high-dimensional (complex) models in simple cage. → Deform coarse mesh, apply deformation to high-res model by a lin. comb. of coarse mesh points.

→ Good for poses, but can be too restr., hard to have direct control

Skeletal Animation: Shape „implies" Skeleton → Animate skeleton, have „Skin" (Mesh) follow the movements

Forward Kinematics: Hierarchy of affine transformations

- Joints: Local coord. Frames
- Bones: Vectors between pairs of joints

↳ Each non-root bone defined in frame of unique parent $p(j)$ → Changes to parent frame affect all children bones

→ Skeleton & Skin designed in Rest (Bind) pose

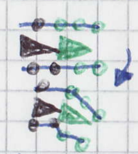
Transf. from frame j to world:

${}_wR_j = {}_wR(0) \dots {}_{p(p(j))}R(p(j))\ {}_{p(j)}R(j)$

→ Starting from root node 0, apply all (relative) transf. to parents of j to go from rel. coord. frame to (absol.) world coordinates

Rigid Skinning: Assign each mesh vertex to closest bone, compute world coordinates according to bone's transform.

→ Unwanted „wrinkles" near joints



→ fine if mesh close to bone or small rotations → Used in old games

Linear Blend Skinning: Assign each mesh vertex to multiple bones, compute world coordinates as convex combination; weights are the influence of each bone of the vertex → Leads to smoother deformation

$$v = \Sigma_j\, \alpha_j\ \underbrace{{}_wR(j)\ {}_w\bar{R}(j)^{-1}\ \bar{v}}$$

Skin vertex — weights — vert. coord. in frame of bone j — vert. in rest pose

Physically Based Animation

Configuration of a system:

$q(t) = (x_0(t), x_1(t), \dots, x_n(t)) \in \mathbb{R}^n$

→ Vector of positions of elements $0 .. n$ at timepoint t

Velocity of points in system $q(t)$

= First derivative $\dot{q}(t)$

Numerical Integration

- Have: $q(0)$ [Pos. at $t=0$] & $\dot{q}(0)$ [Velocity at $t=0$]
- Want: $q(t)$ → Use numerical integration, with $q(t+h) = q(t) + \int_t^{t+h} \dot{q}(t)\,dt$

→ Discrete Approxim. $q_{i+1} = q_i + \Delta q_i$

Rectangle: $\Delta q_i = \dot{q}(t) \cdot h$

Midpoint: $\Delta q_i = \dot{q}(t+h/2) \cdot h$

Trapezoid: $\Delta q_i = \frac{\dot{q}(t) + \dot{q}(t+h)}{2}$

Explicit (Forward) Euler

- $q_{i+1} = q_i + h \cdot \dot{q}_i$ (Appl. Triangle rule)
- \+ Simple & intuitive: „Walk a bit in direction of derivative"
- \- Not very stable (may need very small steps)
- Evaluate derivative $\dot{q}$ at current config.

Implicit (Backward) Euler

- $q_{k+1} = q_k + h \cdot f(q_{k+1})$; $f(q_{k+1})$ = velocity at next timestep, only implicitly!
- \+ Unconditionally stable
- \- Numerical dampening, slow iterations

Symplectic Euler

Update velocity using current config, update config using new velocity

- \+ Easy to implement
- \+ Energy is conserved almost exactly
- \- Only for 2nd order PDEs

Optimization

Gradient Descent $\nabla f(x_1, x_2, \dots x_n)$

- General Update Rule: $x_{k+1} = x_k - \tau \nabla f(x)$, $\nabla f = \begin{bmatrix} f\,dx_1 \\ f\,dx_2 \\ \vdots \\ f\,dx_n \end{bmatrix}$

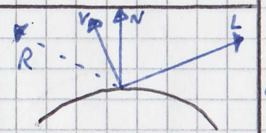
Optimality Conditions

x^* is a local min. of $f(x)$ iff:

- Gradient with respect to $f(x)$ is zero
- Hessian Matrix is pos. definite

Missed Stuff

Phong Shading



- N = Normal Vector; L = Towards Light src; V = Vect. towards Camera.

R = Reflection Vect; $R = 2(L \cdot N) \cdot N - L$ $= (L \cdot N)N + (L \cdot N)N - L$

$I = I_a k_a + I_d k_d (N \cdot L) + I_s k_s (R \cdot V)^\alpha$

ambient + diffuse + specular

Reflection Terms of Phong Shading:

- Ambient, diffuse & specular.
- Reflection vector R only used in specular term

Low-Pass From High-Pass

Low-Pass = Impulse-Kernel − High-Pass

„Img" $*$ $\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$ =

Ideal Low-Pass Filter

- Completely elim. all freq. above cutoff

→ Freq. response = Box (e.g. Sinc)

→ Can cause ringing if applied to img

Kernel Buffer Solution

Example: Kernel has only 3 values. For each pixel, cal. pixel x value (for all 3 values) and store them → When needed in convolution, can just look up → $3N^2$ istead of $9N^2$

Bilateral Filtering $I'(x,y) =$

$\frac{1}{z} \sum f(i,j)\, g(I(x,y) - I(x+i, y+j)) \cdot I(x+i, y+j)$ with $(i,j) \in N(x,y)$ and $g: \mathbb{R} \to \mathbb{R}$ is a gaussian.

⇒ Edge-preserving smoothing; introduces a content-based filter by considering intensity diff. between Neighb.

⇒ Not spatially invariant b/c it is a product of gaussian on dist. & a gaussian on intensity difference

⇒ $z = 1 = \sum_{i,j} f(i,j)\, g(I(x,y) - I(x+i, y+j))$

Normal Mapping: Map „normal" image where each pixel has its own normal

⇒ Looks like shadows are applied

Cubic Hermite Spline p = Points; m = Tangents

$p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (-2t^3 + 3t^2)p_1 + (t^3 - t^2)m_1$

Radon Transformation

Backproj.: Shoot multiple rays at obj. from different angles, determine shape of obj.

Radon Transf. takes f def. on a plane to R_f on 2D space of lines in plane. Val. of a line = line integral of that line trough obj.

CT: f is unknown density, R_f value of tomogr. scan. Inverse of R_f can be used to reconstruct original density from project. data.

Euler Examples

- Spaceship in 2D, m = mass, F = Force forw.

→ State $x = \binom{p}{v}$ → $\dot{p} = v$; $\dot{v} = \ddot{p} = \frac{F}{m}$

→ Expl. Euler: $p_{i+1} = p_i + \Delta t\, v_i$

$v_{i+1} = v_i + \Delta t\, F/m$

→ Symplectic Euler: $v_{i+1} = v_i + \Delta t \frac{F}{m}$

$p_{i+1} = p_i + \Delta t \cdot v_{i+1}$