The Digital Image

Problems: Transmission interference, compression artifacts, spilling, scratches, sensor noise, bad contrast and resolution, motion blur

Pixel: Discrete samples of an continuous image function

Charge Coupled Device (CCD): Has an array of photosites (a bucket of electrical charge) that charge proportional to the incident light intensity during exposure. ADC happens line by line.

Blooming: oversaturation of finite capacity photosites causes the vertical channels to "flood" (bright vertical line)

Bleeding/Smearing: While shifting down, the pixels above get some photons on bright spot with electronic shutters.

Dark Current: CCDs produce thermally generated charge they give non-zero output even in darkness (fluctuates randomly) due to spontanous generation of electrons due to heat → can be avoided by cooling, worse with age

CMOS: Same sensor elements as CCD, but each sensor has its own amplifier → more noise, lower sensitivity

vs CCD: cheaper, lower power, less sensitive, per pixel amplification random pixel access, no blooming, on chip integration

Sampling methods: Cartesian (grid), hexagonal, non-uniform

Quantization: Real valued function will get digital values (integers). A lossy process (original cannot be reconstructed). Simple version: equally spaced $2^{b=\#bits}$ levels

Bilinear Interpolation:

$$f(x,y) = (1-a)(1-b)f[i,j] + a(1-b)f[i+1,j] + ab f[i+1,j+1] + (1-a)b f[i,j+1]$$

Resolution: Image resolution (cropping), geometric resolution (#pixels per area), radiometric resolution (#bits per pixel, color)

Image noise: commonly modeled by additive Gaussian noise: $I(x,y) = f(x,y) + c$, poisson noise (shot noise for low light, depends on signal & apature time), multiplicative noise: $I = f + fc$, quantization errors, salt-and-pepper noise

SNR or peak SNR is used as an index of image quality

$c \sim \mathcal{N}(0,\sigma^2)$, $p(c) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(c-\mu)^2}{2\sigma^2}}$, $p(k) = \frac{\lambda^k e^{-\lambda}}{k!}$, SNR: $s = \frac{F}{\sigma}$ where $F = \frac{1}{XY}\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} f(x,y)$

Color cameras: prism (need 3 sensors and good alignment, filter mosaik (coat ■ directly on sensor), wheel (multiple filters in front of same sensor), new CMOS sensor (layers that absorb color at different depths → better quality)

Image Segmentation

Complete segmentation: finite set of non-overlapping regions that cover the whole image. $I = \bigcup_{i=1}^{N} R_i$ and $R_i \cap R_j = \emptyset \ \forall i,j: i \neq j$

Thresholding: simple segmentation by comparing greylevel with a threshold to decide if in or out.

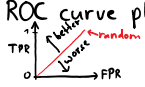
Chromakeying: When planning to segment, use special background color. Problems: variations due to lighting, noise,... mixed pixels (hard α-mask does not work) $I_\alpha = |I - g| > T$

Receiver Operating Characteristic (ROC) Analysis:

ROC curve characterizes performance of binary classifier

Classification errors: False negatives (FN), false positives (FP)

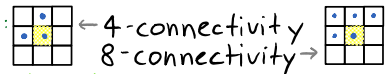
ROC curve plots TP fraction TP/(TP+FN) vs FP fraction FP/(FP+TN)



Operating points: choose point with gradient

$\beta = \frac{N}{P}\frac{V_{TN} + C_{FP}}{V_{TP} + C_{FN}}$ V: Value C: Cost

Pixel connectivity: (also regions if x-connected)



Connected component raster scanning: scanning row by row, if foreground & label if connected to other label else give new label. Second pass to find equivalent labels.

Improve: when region found, follow border, then carry on (contour-based method)

Region growing: Start with seed point or region, add neighboring pixels that satisfy a criteria defining a region until we include no more pixels.

Seed region: by hand or automatically by conservative thresholding

Inclusion criteria: greylevel thresholding, greylevel distribution model (include if $(I(x,y)-\mu^2)^2 < (n\sigma)^2$ and update μ, σ after each iteration) color or texture information

Snakes: active contour, a polygon and each point moves away from seed while criteria is met, can have smoothness constraint

Iteratively minimize energy function $E = E_{tension} + E_{stiffness} + E_{image}$

Background subtraction: Simple: $I_\alpha = |I - I_{bg}| < T$

better: $I_\alpha = \sqrt{(I-I_{bg})^T \Sigma^{-1} (I-I_{bg})}$ where Σ is the background pixel appearance covariance matrix, computed seperately for each pixel.

Morphological operators

Logical transformations based on comparison of neighboring pixels: **erode:** delete FG pixels with 8-connected BG pixels, **dilate:** every BG pixel with 8-connected FG pixel make a FG pixel.

Uses: smooth regions, remove noise and artifacts.

Image Filtering

Modify the pixels of an image based on some function of the local neighborhood of the pixels. If sum greater 1 get brighter, if smaller darker.

seperable: if a kernel can be written as a product of two simpler filters: e.g. sobel: $\begin{bmatrix}1\\2\\1\end{bmatrix} * A = \begin{bmatrix}1\\2\\1\end{bmatrix} * [1\ 0\ -1] * A$

↳ computationally faster filter P×Q, image N×M: (P+Q)NM instead of PQNM

shift invariant: Doing the same thing (applying the same function) over all pixels (in the formula below if K does not depend on x, y)

linear: linear combination of neighbors can be written as: $I'(x,y) = \sum_{(i,j)\in N(i,j)} K(x,y;i,j)\, I(x+i, y+j)$ (I': output, N(i,j): neighborhood, K: Kernel, I: Input)

Filter at edges: clip filter (black), wrap around, copy edge, reflect across edge, vary filter near edge

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

e.g. template matching: search for best match by minimizing mean squared error or maximizing area correlation. (remove mean to avoid bias: from filter & from image)

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

(same correlation, but with kernel reversed $\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} \to \begin{bmatrix}9&8&7\\6&5&4\\3&2&1\end{bmatrix}$)

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

↳ if $K(i,j) = K(-i,-j) \Rightarrow$ correlation = convolution

Continuous: $(f*g)(t) = \int_{-\infty}^{\infty} f(\tilde{f})\, g(t-\tilde{f})\, d\tilde{f} = \int_{-\infty}^{\infty} f(t-\tilde{f})\, g(\tilde{f})\, dt$

Kernels

Box filter: all same values, normalized so sum = 1

Gaussian kernel: $G_\sigma = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{(x^2+y^2)}{2\sigma^2}}$ (is seperable, e.g. $\sigma = 1$)

↳ Rotationally symmetric, has single lobe, single lobe in frequency domain, simple relationship to σ, easy to implement efficiently, neighbors decrease monotonically, no corruption from higher freq.

Subtracting one from central element of low-pass filter gives high-pass filter with inverted sign, because:

$(1-\delta) * a = f*a - \delta*a = f*a - a = -(a - (f*a))$ (e.g. filter $\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$ use at (x_0,y_0))

Band pass filter: ▣ do LPF and HPF with cutoffs $D_{LP} < D_{HP}$.

Band reject filter: ▣ do LPF and HPF with cutoffs $D_{LP} > D_{HP}$

Image sharpening: increases high frequency components to enhance edges: $I' = I + \alpha|K * I|$ (K: high pass filter, α: scalar $\in [0,1]$)

Computer Vision Tasks: Single object: Classification Classification + Localization (bounding box) Multiple objects: Object detection, Instance segmentation

Features

Edge Detection

How to tell if there is an edge?

Local maxima of the first derivative and the zero crossings of the second derivative.

Edge detection filters:

image gradients 1st deriv.: $\begin{bmatrix}-1&0&1\\-1&0&1\\-1&0&1\end{bmatrix}$ Prewitt ← $\begin{bmatrix}-1&-1&-1\\0&0&0\\1&1&1\end{bmatrix}$ Sobel → $\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}$ $\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}$ $\begin{bmatrix}0&1\\-1&0\end{bmatrix}$ $\begin{bmatrix}1&0\\0&-1\end{bmatrix}$ Roberts

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

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

Laplacian operator: detect discontinuities by considering second derivative: $\begin{bmatrix}0&1&0\\1&1&1\\0&1&0\end{bmatrix}$ or $\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$ are discrete space approximations. Is isotropic (rotationally invariant), zero-crossings mark edge locations. Sensitive to fine details and noise → blur image first (LoG)

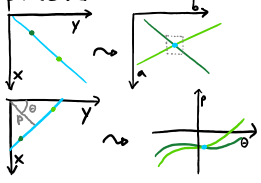
Laplacian of Gaussian (LoG): convolve gaussian blurring and laplacian operator into LoG operator (cheaper) $LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1-\frac{x^2+y^2}{2\sigma^2}\right]e^{-\frac{(x^2+y^2)}{2\sigma^2}}$

Canny Edge Detector: 5 steps

1) Smooth image with a Gaussian filter
2) Compute gradient magnitude and angle
3) Apply non-maxima suppression to gradient magnitude image. Quantize edge normal to one of four directions: horizontal, +45°, vertical, -45°. If M(x,y) smaller than either of its neighbors in edge normal direction suppress, else keep
4) Double thresholding to detect strong and weak edge pixels
5) Reject weak edge pixels not connected with strong edge pixels

Hough Transform

Fitting a straight line to a set of edge pixels



$y = ax + b$ detect where many intersections occur.

Problem: "Infinite slope", vertical lines cannot be detected
-cannot tell line from segment

Alternative parameterization: $x\cdot\cos\theta + y\cdot\sin\theta = \rho$

For circles: if r known: calculate circles with radius r around edge pixels → where lots of them meet is the center of a circle.
else: use 3D hough transform with parameters (x_0, y_0, r)

Corner Detection

Edges are only well localized in one direction → detect corners
Desirable properties: Accute localization, invariance against shift, rotation, scale, brightness change, robust against noise, high repeatability

Linear approximation for small $\Delta x, \Delta y$: (Taylor)
$f(x+\Delta x, y+\Delta y) \approx f(x,y) + f_x(x,y)\Delta x + f_y(x,y)\Delta y$

Local displacement sensitivity (Harris corners):

$$S(\Delta x, \Delta y) = \begin{pmatrix}\Delta x & \Delta y\end{pmatrix}\underbrace{\left(\sum_{(x,y)\in window}\begin{bmatrix} f_x^2(x,y) & f_x(x,y)f_y(x,y)\\ f_x(x,y)f_y(x,y) & f_y^2(x,y)\end{bmatrix}\right)}_{:=M}\begin{pmatrix}\Delta x\\ \Delta y\end{pmatrix} \approx \text{SSD}$$

Find points where $\min \Delta^T M \Delta$ is large for $\|\Delta\|=1$ i.e. maximize the eigenvalues of M!
↳ $C(x,y) = \det(M) - k\cdot(\text{trace}(M))^2 = \lambda_1\lambda_2 + k\cdot(\lambda_1+\lambda_2)$ Harris cornerness: Measure of cornerness

Robustness of Harris corner detector: Invariant to brightness offset, invariant to shift and rotation, but not to scaling!

$\lambda_1 \gg \lambda_2$ ~edge, λ_1 and λ_2 large ~corner, else ~flat region.
not scale invariant: ◻ ~corner ◇ ~edge (might be detected differently)

Overcome issues: look for strong DoG response or consider local maxima in position and scale space, Gaussian weighing

Lowe's SIFT features

Look for strong responses of difference of Gaussians (DoG) filter, only look at local maxima in both position and scale.
DoG: $DOG(x,y) = \frac{1}{k}\cdot e^{-\frac{x^2+y^2}{(k\sigma)^2}} - e^{-\frac{x^2+y^2}{\sigma^2}}$, e.g. $k=\sqrt{2}$
Orientation: create histogram of local gradient directions computed at selected scale, assign canonical orientation at peak of smoothed histogram. Get a SIFT descriptor (threshold image gradients are sampled over 16x16 array of locations in scale space) and do matching with these. Invariant to scale, rotation, illumination and viewpoint.

Fourier Transform

Aliasing: Happens when undersampling e.g. taking every second pixel, else characteristic errors appear: typically small phenomena look bigger, fast phenomena look slower.
e.g. wagon wheels backwards in movies, checkerboards misrepresented

Fourier Transform: Represent function on a new basis with basis elements $e^{-i2\pi(ux+vy)} = \cos(2\pi(ux+vy)) + i\cdot\sin(2\pi(ux+vy))$
$F(u) = \int_{-\infty}^{\infty} f(x)e^{-2\pi jfx}dx$, **2D:** $F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-2\pi j(ux+vy)}dxdy$
For images: transformed image → $F = U\cdot f$ ← vectorized image, U: Fourier matrix
For discrete: $F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-2\pi j\left(\frac{x}{M}u + \frac{y}{N}v\right)}$
1D-periodic function: $f(t) = \sum_{n=-\infty}^{\infty} c_n\cdot e^{\frac{2\pi jnt}{T}}$ where $c_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)e^{-\frac{2\pi jn}{T}}dt$

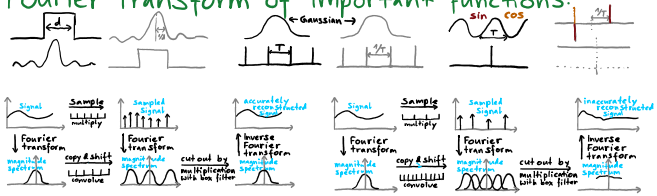
Properties of Fourier transform

Linearity: $F(ax(t)+by(t)) = aX(t)+bY(t)$
time-shift: $\mathcal{F}(x(t\pm t_0)) = X(f)e^{\pm 2\pi i f t_0}$
Convolution theorem: $F\cdot G = \mathcal{F}(f*g)$, $F*G = \mathcal{F}(f\cdot g)$
$\int_{-\infty}^{\infty} f(t)\delta(t-n\Delta T)e^{-2\pi jn\nu t}dt = f(n\Delta T)e^{-2\pi jn\omega n\Delta T}$ $\delta(u,v) = \int_{\mathbb{R}^2} e^{-2\pi i(ux+vy)}dxdy$
$\delta * s = s$ **unit pulse:** $\int_{-\infty}^{\infty} f(t)\delta(t-t_0)dt = f(t_0)$ also for discrete/2D
frequency shift: $\mathcal{F}^{-1}(X(j\omega\pm\omega_0)) = x(t)e^{\mp j\omega_0 t}$ **Symmetry:** $F(x(t)) = 2\pi X(-j\omega)$
time and frequency scaling: $F(x(at)) = \frac{1}{a}X\left(\frac{\omega}{a}\right)$
Phase: $\phi(F) = \tan^{-1}\left(\frac{\text{Im}(F)}{\text{Re}(F)}\right)$ **Magnitude:** $|F| = \sqrt{\text{Re}(F)^2+\text{Im}(F)^2}$

Sampling: A sampling function s(t) which is an impulse train with period T and its Fourier transform S(f):
$S(t) = \sum_{n=-\infty}^{\infty}\delta(t-nT)$ $S(f) = \frac{1}{T}\sum_{n=-\infty}^{\infty}\delta(f-\frac{n}{T})$ where $\delta(\cdot)$ is the Dirac-delta function
A continuous signal can be sampled by multiplying with $s(t)$: $x_s(t) = x(t)s(t)$. To compute the Fourier transform of $x_s(t)$, we can use the convolution theorem:
$\mathcal{F}(x_s(t)) = X(t)*S(t) = \frac{1}{T}\sum_{n=-\infty}^{\infty}\delta(f-\frac{n}{T})*X(t) = \frac{1}{T}\sum_{n=-\infty}^{\infty}X(f-\frac{n}{T})$
Sampling in 2D: $\text{sample}_{\Delta}(f(x,y)) = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f(x,y)\cdot\delta(x-i,x-j) = f(x,y)\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,x-j)$

↳DFT: $F(f(x,y)\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,x-j)) = F(f(x,y))*F(\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,x-j)) = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}F(u-i,v-j)$
Dirac delta: $\delta(x) = \begin{cases}\text{undefined} & x=0\\ 0 & \text{for } x\neq 0\end{cases}$ s.t. $\int_{-\infty}^{\infty}\delta(x)dx = 1$
sifting property: $\int_{-\infty}^{\infty} f(x)\delta(x-a)dx = f(a)$
Dirac comb: $\text{Ш}_T(x) = T\sum_{i=-\infty}^{\infty}\delta(x-iT)$ ⇒ sampling = product with this
Box filter: $h(x) = \begin{cases}1/T & |x|\leq T/2\\ 0 & \text{otherwise}\end{cases}$ $f_{\text{recon}} = (h*g)(x) = T\int_{-\infty}^{\infty}h(y)\sum_{i=-\infty}^{\infty}f(iT)\delta(x-y-iT)dy$
Triangle filter: $h(x) = \begin{cases}(1-|x|/T)/T & |x|\leq T\\ 0 & \text{otherwise}\end{cases}$

Fourier transform of important functions:



Nyquist sampling theorem: The sampling frequency must be at least twice the highest frequency $\omega_s \geq 2\omega$. If not the case: bandlimit before with low-pass filter. Perfect reconstruction: $\text{sinc}(x) = \sin(\pi x)/\pi x$

Why should this hold? (function f(t), sampling function $S_{\Delta t}(t)$ with sampling frequency ω_s). Fourier transform of the sampled function can be derived as $\tilde{F}(u) = F(f(t)\cdot S_{\Delta t}(t)) = F(u) * S_{\Delta t}(\omega) = \int_{-\infty}^{\infty}F(\tilde{f})S_{\Delta t}(\omega-\tilde{f})d\tilde{f}$
$= \int_{-\infty}^{\infty}F(\tilde{f})\frac{1}{\Delta T}\sum_{n=-\infty}^{\infty}\delta(\omega-\tilde{f}-\frac{n}{\Delta T})d\tilde{f} = \frac{1}{\Delta T}\sum_{n=-\infty}^{\infty}F(\omega-n\omega_s)$. If we want to reconstruct the signal f(t) from F and $S_{\Delta T}$, $F(\omega)$ cannot overlap with its neighbors $F(\omega-\omega_s)$ and $F(\omega-\omega_s)$. Thus ω_s should be larger than ω_n (highest frequency of $f(t)$)

Image restoration problem: $f(x) \to \boxed{h(x)} \to g(x) \to \boxed{\tilde{h}(x)} \to f(x)$
The "inverse" kernel $\tilde{h}(x)$ should compensate $h(x)$. May be determined by: $\mathcal{F}(\tilde{h})(u,v)\cdot\mathcal{F}(h)(u,v) = 1$. **Problems:** Convolution with kernel k may cancel out some frequencies & noise amplification.
Avoid: Regularization: $\tilde{\mathcal{F}}(\tilde{h})(u,v) = \mathcal{F}(h)/|\mathcal{F}(h)|^2+\varepsilon$ (avoid singularities)

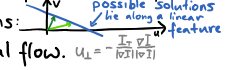
Optical Flow

Apparent motion of brightness patterns (projection of 3D velocity vectors on I)
Use extracted feature points and compute their velocity vectors
Problem: cannot distinguish motion from changing lighting!
also: estimate observed projected motion field (normal flow) not always well defined

Key assumptions: Brightness constancy: projection of the same point looks the same in every frame. Small motion: points do not move far. Spatial coherence: points move like their neighbors.

Brightness constancy constraint: $I(x,y,t-1) = \underbrace{I}_{\text{Intensity}}(x+u, y+v, t)$
Small motion → can linearize with Taylor expansion:
$I(x,y,t-1) \approx I(x,y,t-1) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + e$ with very small δt:
$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} \approx 0$ or shorthand $I_x\cdot u + I_y\cdot v + I_t \approx 0$
↳move I_t on one side, vectorize unknowns. For LK, sum up over a window of pixels

Apature problem: 1 equation, 2 unknowns:



↳cannot determine exact location, take normal flow. $u_\perp = -\frac{I_t}{\|\nabla I\|}\frac{\nabla I}{\|\nabla I\|}$

Horn & Schunck algorithm: Add additional smoothness constraint: $e_s = \iint((u_x^2+u_y)^2+(v_x^2+v_y^2))dxdy$ (if close should be almost parallel) besides OF constraint $e_c = \iint(I_x u + I_y v + I_t)^2 dx dy$. Minimize $e_s + \lambda e_c$

Lukas-Kanade: Assume same displacement for N×M (e.g. 5×5) window
↳linear least squares problem: $\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ \vdots & \vdots \\ I_x(p_{nm}) & I_y(p_{nm}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\begin{bmatrix} I_t(p_1) \\ I_t(p_2) \\ \vdots \\ I_t(p_{nm}) \end{bmatrix} \rightarrow \begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$
When solvable? A^TA invertible, eigenvalues λ_1, λ_2 large, λ_1/λ_2 small
Errors: motion is larger(r than a pixel) → iterative refinement and coarse-to-fine estimation. A point does not move like its neighbors →motion segmentation. Brightness constancy does not hold: →exhaustive neighborhood search with normalized correlation.
KLT feature tracker: to find patches where LSE well-behaved → LK-flow
Iterative refinement: Estimate velocity, warp using estimate, refine,...
Coarse-to-Fine Estimation: Image Pyramid. Start small, compute OF, rescale, take larger and initialize with last estimate
Applications: Image stabilization (get flow between two frames and warp image using same OF for all pixels s.t. OF close to 0) frame interpolation, video compression, object tracking, motion segmentation

Affine motion: $I_x(a_1+a_2x+a_3y)+I_y(a_4+a_5x+a_6y)+I_t \approx 0$

extension to planar perspective: 
complexity ↑ stability ↓
SSD tracking: For large displacements: match template against each pixel in small area around, match measure can be (normalized) correlation or SSD, choose max. as match (sub-pixel also possible)
Bayesian Optical Flow: Some low-level motion illusions can be explained by adding an underlying model to LK-tracking e.g. brightness constancy with noise.

Video Compression

Interlaced video format: 2 temporally shifted half images →increase frequency, decrease spatial resolution →not progressive
Lossy video compression: take advantage of redundancy spatial correlation between pixels, temporal correlation between frames →basically drop perceptually unimportant details.
With optical flow: encode optical flow based on previous frame can cause blocking artifacts, does not work well for lots of movement, fast movement and scene changes.
If temporal redundancy fails → use motion-compensated prediction
Types of coded frames:
- **I-frame:** intra-coded frame, coded independently of all others
- **P-frame:** predictively coded frame, based on previously coded frame
- **B-frame:** bi-directionally coded frame, based on previous & future

Block-Matching Motion Estimation:
Is a type of temporal redundancy reduction.
Motion Estimation Algorithm (ME)
1) Partition frame into blocks (e.g. 16x16 pixels)
2) For each block, find the best matching block in reference frame

Metrics for best match: sum of differences or squared sum of diff.
Candidate blocks: All blocks in e.g. 32x32 pixel area
Search strategies: Full search, partial (fast) search

Motion Compensation Algorithm (MC)
Use the best matching of reference frame as prediction of blocks in current frame.
↳ gives motion vectors & MC prediction error or residual (encode with conventional image coder)
Motion vector: relative horizontal & vertical offsets of a given block from one frame to another. Not limited to integer-pixel offsets, can use half-pixel ME to capture sub-pixel motion.
Half-pixel ME (coarse-fine) algorithm
1) coarse step: find best integer MV
2) fine step: refine by spatial interpolation and best-matching
Advantages and disadvantages
+ good, robust performance, one MV per block → useful for compression, simple periodic structure (GoP)
− assumes translational motion (fails for complex motion →code these frames/blocks without prediction) produces blocking artifacts.
MPEG-GoP: IBBPBBPBBI dependencies between frames

Scalable Video Coding: Decompose video into multiple layers of prioritized importance: e.g. temporal scalability: Include B-frames or not. Spatial scalability: Base resolution + upsampling difference. SNR Scalability: Base with coarse quantizer + finer quantizer
Benefits: Adapting to different bandwidths, facilitates error resiliency by identifying more and less important bits.

Unitary Transforms

Vectorization: interpret image as vector row-by-row: $\begin{matrix}1&2\\3&4\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix}$
linear image processing: can be written as $\vec{g} = H\vec{f}$
Image collection (IC): $F = [f_1 f_2 \dots f_n]$
Autocorrolation matrix: $R_{ff} = \frac{F \cdot F^H}{N}$ its Eigenvector with largest Eigenvalue is direction of largest variance among pictures.
Unitary transform: for transform A iff $A^{-1} = A^H$ (if real-valued → orthonormal)
energy conservation: $\|\vec{c}\|^2 = \vec{c}^H\vec{c} = \vec{f}^H A^H A \vec{f} = \|\vec{f}\|^2$ → every unitary transform is a rotation + sign flip. Lengths conserved
↳energy conserved but often unevenly distributed among coeff.

Karhunen-Loeve Transform: Same as PCA. Order by decreasing eigenvalues. **Energy concentration property:** no other unitary transform packs as much energy in the first J coefficients (for arbitrary J) and mean squared approximation error by choosing only first J coefficients is minimized.
Optimal energy concentration of KLT: consider truncated coefficient vector $\vec{b} = I_J \vec{c}$ $I_J = \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & 0 \end{bmatrix}$. Energy in first J coefficients for an arbitrary transform A: $E = Tr(R_{bb}) = Tr(I_J R_{cc} I_J) = Tr(I_J A R_{ff} A^H I_J) = \sum_{k=0}^{J-1} a_k^T R_{ff} a_k^*$ where a_k^T is k-th row of A. Lagrangian cost function to enforce unit-length basis vectors: $L = E + \sum_{k=0}^{J-1} \lambda_k (1 - a_k^T a_k^*) = \sum_{k=0}^{J-1} a_k^T R_{ff} a_k^* + \sum_{k=0}^{J-1} \lambda_k (1 - a_k^T a_k^*)$
Differentiating L with respect to a_j: $R_{ff} a_j^* = \lambda_j a_j^* \; \forall j < J$. (necessary condition)

Simple recognition: SSD between images, best match wins
↳very expensive, since need to correlate with every image

Principle Component Analysis (PCA)

Steps: standardize data, get Eigenvectors and values from covariance matrix or do SVD, sort Eigenvalues and vectors in descending order get j largest components, construct projection matrix from selected j Eigenvectors, transform dataset by multiplying with projection matrix.
Draw PCA: 

where biggest variance is perpendicular to that
Uses of PCA: lossy compression by keeping only the most important k components. Face recognition (eigenfaces) and face detection.

Eigenspace matching: Do PCA (with mean subtraction) and get closest rank-k approximation of database images (eigenfaces) For a new query: normalize, subtract mean (of database) project to subspace, then do similarity matching with eigenfaces. (multiply with U_k^T)
Limitations: Lighting differences much larger than face diff.

Fischerfaces: Find directions where ratio between within individual variance is maximized. Linearly project to basis where dimension with good signal: noise ratio is maximized. $W_{opt} = \arg\max \left(\frac{|det(W S_B W^H)|}{|det(W S_W W^H)|}\right)$, $R_B = \sum_{i=1}^{c} R_B \sum_{i=1}^{c} N_i (\vec{\mu}_i - \vec{\mu})(\vec{\mu}_i - \vec{\mu})^H$, $R_W = \sum_{i=1}^{c} \sum_{f \in class_i} (f_i - \mu_i)(f_i - \mu_i)^H$
Fischer linear discriminant analysis (LDA): maximize between class scatter, while minimizing within class scatter.

JPEG Compression: Divide image into 8x8 block:
8x8 block → DCT → Quantization Matrix → DPCM → DC Huffman → ZigZag Scan → AC Huffman
DC: First coefficient (general intensity) **ZigZag:**  top left: high freq, bottom right: low fr.
Quantization Table: Divide by this value, round to nearest integer.
Why DCT? Compared to DFT uses only real values, easier to compute

Pyramids and Wavelets

Scale-space representations: From an original signal f(x) generate a parametric family of signals $f^+(x)$ where fine-scale information is sucessively suppressed e.g. successive smoothing or image pyramids (smooth & downsample)
Applications: Search for correspondance (look at coarse scale, then refine with finer scale, edge tracking coarse-to-fine estimation

control of detail and computational cost (e.g. textures)
Example: CMU face detection: need different scales for template to match

Gaussian Pyramid: Image pyramid with Gaussian for smoothing
Laplacian Pyramid: preserve difference between upsampled Gaussian pyramid level and Gaussian pyramid level. Like a band-pass filter - each level represents spatial frequencies that are largely unrepresented at other layers. → Compression

1D-Discrete Wavelet transform: recursively application of a two-band filter bank to the low pass band of the previous stage yield octave band splitting
Haar transform: has two major sub-operations:

- scaling captures info. at different frequencies
- translation captures info. at different locations

Textures

Appearance Modelling: We have multi-view videos and the shape and want to compute the appearance
Appearance space $\mathbb{T}$, surface S, find $\Phi := \mathbb{T} \rightarrow S$ (appearance representation)
Challenges: Noisy captured images, visual redundancy over space, callibration inaccuracies, reconstruction inaccuracies, occlusions, visual redundancy over time geometric noise (reconstruction noise & calibrating noise)
Blending: Naïve approach, just see where it gets projected (on multi-view video) get RGB values and project back. Limitation: Geometric inaccuracies lead to strong blurring effects.
Multi-view texture estimation: Like blending, but pick the best value among views. Limitation: visual redundancy not exploited.
Super resolution: Compute the value that is close to observations. Limitation: Geometric inaccuracies not compensated
Image based rendering: Optical flow to correct misalignments. Limitation: view-dependent appearance representation.
Temporal multi-view case: exploit temporal visual redundancy
Limitation: spatial visual redundancy not exploited

Goals of static multi-view case

- provide a compact view-independent representation
- exploit visual redundancy to increase quality
- account for geometric inaccuracies

Generative model: appearance + model → multi-view images

Image formation model: $I_i = S \cdot K \cdot F_{w_i} \cdot P_i \cdot T + N_i$
S: downsampling, K: Gaussian kernel, F_{w_i}: optical flow from P_i to new, P_i: projection onto model, T: appearance map, N_i: thermal noise
⇒ find best T that explains the input I_i

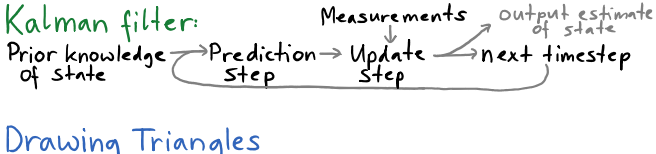
Augmented reality

Ingredients: (in-air) display, tracking (user/camera), scene reconstruction, user input → all in realtime
Tracking technologies: **Active:** mechanical, magnetic, ultrasonic, GPS, Wifi, Cell location. **Passive:** inertial sensors (compass, accelerometer, gyro), vision based (marker based, natural feature tracking) **Hybrid:** combined sensors (e.g. vision + inertial)
Marker-based tracking: simplest form of tracking, with key idea: rectangle provides 4 corner points, which is enough to estimate full pose.
Scene reconstruction: **Localization:** camera pose estimation **Mapping:** reconstruct environment geometry.

Simultaneous Localization and Mapping (SLAM): the process by which we build a map of the environment, at the same time, use this map to compute the camera location. **Localization:** inferring location given a map
Mapping: inferring a map given a location.
⇒ chicken-or-egg problem, fundamental for autonomous robots
Full SLAM: Estimate entire path and map
Online SLAM: Estimate most recent pose and map

SLAM uncertainty: robot tracks object to small uncertainty area, based on that robot moves to a larger uncertainty area & tracks new objects with even larger uncertainty. To reduce uncertainty can observe features whose location is relatively well known, e.g. features observed before (→ loop closure detection)

SLAM algorithms: Extended Kalman Filter SLAM, Particle Filter SLAM, Graph-Based SLAM

Kalman filter:


Drawing Triangles

Rasterization: converting continuous object to a discrete representation on a pixel grid (raster grid)
Diamond rule: ◻ if it intersects gray line → color in
Incremental line rasterization: Line from (u1, v1) to (u2, v2)
slope: $s = (v2-v1)/(u2-u1)$. e.g. $u1 < u2$, $v1 < v2$ and $0 < s < 1$:
go from left (u1) to right (u2), increment v1 by s every iteration and color in pixel at (u, round(v)).
Coverage: What pixels does a triangle overlap
$\text{coverage}(x,y) = \begin{cases} 1 & \text{if the triangle contains point } (x,y) \\ 0 & \text{otherwise} \end{cases}$
Occlusion: What triangle is closest to the camera

Supersampling: Increase density of sampling coverage signal to >1 sample per pixel then do **resampling** e.g. at display's pixel resolution → anti-aliased image

Point-in-triangle test: Points: $P_i = (x_i, y_i)$, $dX_i = x_{i+1} - x_i$, $dY_i = y_{i+1} - y_i$
Test all three equations: $E_i(x,y) = (x - x_i)dX_i - (y - y_i)dY_i$ and test for:
$=0$ point on edge, >0 point outside edge, <0 point inside edge
$\text{inside}(x,y) = E_0(x,y) < 0 \;\&\&\; E_1(x,y) < 0 \;\&\&\; E_2(x,y) < 0$
Incremental triangle traversal: Instead of testing all points on screen, traverse incrementally. backtrack, zig-zag, Hilbert/Morton curves

Transforms

Linear maps: if for all u, v, a: $f(u+v) = f(u) + f(v) \wedge f(au) = a f(u)$
Linear transform: if it can be expressed as $f(u) = \sum_{i=1}^{m} u_i a_i$ with fixed vectors a_i. → changing coordinate frames
benefits: computationally fast, very powerful, can approximate all maps over a short distance (Taylor's theorem), composition of linear transforms is linear. (transforms A, B → (A·B)·x first B, then A)

Homogeneous coordinates: Lift space to one higher subspace and set its coordinate to 1.
benefits: some non-linear transforms (e.g. affine, perspective projection) to be expressed as matrix-vector operations.
Also vectors ($\omega = 0$) distinguishable from points ($\omega = 1$).
Rotation: $R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$ $R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$ $R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Translation: $T_b = \begin{bmatrix} 1 & & & b_x \\ & 1 & & b_y \\ & & 1 & b_z \\ & & & 1 \end{bmatrix}$ needs to be in 3D-H coordinates to be a linear transform!
Scale: $S_s = \begin{bmatrix} s_x & & \\ & s_y & \\ & & s_z \end{bmatrix}$ **Shear:** $H_{x,d} = \begin{bmatrix} 1 & d_y & d_z \\ & 1 & \\ & & 1 \end{bmatrix}$ (in x, base on y and z position)
Rotation: about arbitrary axis u $\begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$
Map Plane: Given plane $ax + by + cz + d = 0$ and transformation matrix M, normal vector is $(a,b,c)^T$ and new normal vector is $(M^{-1})^T n$. Find new d by transforming point and solving for d.

Quaternions: $z = a + ib + jc + kd$
Properties: $i^2 = j^2 = k^2 = -1$ $ki = j$ $ijk = -1$
$ij = k$ $jk = i$ $ji = -k$ $kj = -i$ $ik = -j$
Addition: $(a_1 + b_1 i + c_1 j + d_1 k) + (a_2 + b_2 i + c_2 j + d_2 k)$
$= (a_1 + a_2) + (b_1 + b_2)i + (c_1 + c_2)j + (d_1 + d_2)k$
Norm: $\|z\| = \sqrt{a^2 + b^2 + c^2 + d^2}$ **unit form:** $z / \|z\|$
scalar/vector form: $z = S + V$ (S is scalar, V is vector)
Multiplication: $z_1 = S_1 + V_1$, $z_2 = S_2 + V_2$
$z_1 z_2 = S_1 S_2 - V_1 \cdot V_2 + S_1 V_2 + S_2 V_1 + V_1 \times V_2$
Conjugate: $z = S + V$, $\bar{z} = S - V$, $z\bar{z} = \|z\|^2$, $z^{-1} = \bar{z} / \|z\|^2$
Rotation: Vector $p = (x,y,z)^T$, rotate around u by θ:

1.) $[x,y,z]^T \longrightarrow$ Quaternion $p = 0 + xi + yj + zk$
2.) compute $q = \cos(\theta/2) + \sin(\theta/2)u$ (u is unit vector)
3.) compute $q^{-1} = \cos(\theta/2) - \sin(\theta/2)u$ ($\bar{q} = q^{-1}$ since q unit quat)
4.) $p' = qpq^{-1}$ (p' is rotated vector p)

Why use quaternions? efficient implementation, easy interpolation, no gimble lock

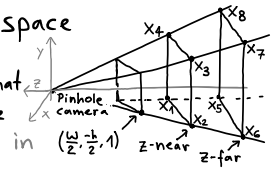
Transform geometry and textures

Basic perspective projection: Assumption: pinhole camera at (0,0,0) looking down z. Input: point in 3D-H: $x = (x_x, x_y, x_z, 1)$. Output: perspective projected point $p_{pp} = (x_x/x_z, x_y/x_z)$

$P = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \end{bmatrix}$ first apply P, then project to 2D-H (just drop z) then transform to 2D (with homogenous divide)

Orthographic projection: simple way to project things. $Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

View frustrum: region in space that will appear on screen.
→ we want a transformation that transforms this into a unit cube (computing screen coordinates in that space is then trivial)



1) apply perspective transform and normalize z-coord
2) rescale x and y components s.t. frustrum maps to unit cube.

$P = \begin{bmatrix} \frac{h}{r} & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & \frac{z_{far}+z_{near}}{z_{near}-z_{far}} & \frac{2 \cdot z_{far} \cdot z_{near}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$ r = aspect ratio = width/height
θ: field of view in y-direction
s.t. $h = \tan(\theta/2)$, $f = 2 \cdot \cot(\theta/2)$

Ways to encode geometry:
explicit: points are given directly. ($f: \mathbb{R}^2 \to \mathbb{R}^3; (u,v) \to (x,y,z)$)
↳ good for sampling, bad for inside/outside tests
implicit: points are not known directly, but satisfy some relationship → good for inside/outside tests, compact description, other queries (e.g. distance to surface) easy, for simple shapes exact description/no sampling error, easy to handle changes in topology, bad for sampling, very difficult to model complex shapes

Examples of implicit geometry:
Algebraic surfaces: surface is zero set of polynomial in x,y,z
Constructive solid geometry: build complicated shapes via Boolean operations. **Blobby surfaces:** gradually blend surfaces together (levels of sum of gaussians). **Blending distance functions:** a distance function gives distance to closest point on object. **Level set methods:** store a grid of values approximating function. **Fractals and L-systems:** no precise definition, structures that exhibit self-similarity, details at all scales.

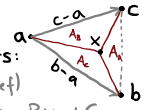
self-similarity, details at all scales.

Examples of explicit geometry:
Point cloud: list of points (x,y,z), often augmented with normals can represent any geometry, need large dataset, hard to do processing/simulation, hard to draw if undersampled. **Polygonal mesh:** store vertices and polygons, easier processing simulation, more complicated DS, most common. **Triangle mesh:** store vertices as triplets (x,y,z), triangles as triplets of indices (i,j,k). **Subdivision surfaces:** Smooth out a control curve, insert new vertex at each edge midpoint and update vertex positions according to fixed rule.

Polygonal Mesh: Set of connected polygons where every edge belongs to at least one polygon and the intersection of two polygons either empty, a vertex or an edge.
Manifolds: surface homeomorphic to a disk, closed manifolds divides space into two.

Sources of geometry: Acquired real-world objects via 3D scanning, digital 3D modelling

Barycentric interpolation: color of x is an affine combination of vertex colors:



$x_{color} = \alpha a_{color} + \beta b_{color} + \gamma c_{color}$ with $\alpha + \beta + \gamma = 1$ (by def)
For any surface attribute, value at x is: $f_x = Ax_x + Bx_y + C$.
Get A,B,C by plugging in corner points attributes (f_x) and coords (x_x, x_y) for all three corners, then solve LSE for A,B,C.
↳ problem: perspective-incorrect interpolation

Perspective-correct interpolation: attribute values f vary linearly across triangle in 3D. Due to perspective projection, f/z varies linearly in screen coordinates, but not f directly. Basic recipe:
1) Compute depth z at each vertex. 2) Evaluate $z' = 1/z$ and $P = f/z$ at each vertex. 3) Interpolate z' and P using standard 2D barycentric interpolation. 4) At each fragment divide interpolated P by interpolated z' to get f.

Texture mapping: describing surface material properties
Normal mapping: Use texture value to perturb surface normal to give appearance of a bumpy surface. (problem: smooth silhouette and smooth shadow, for that need high-resolution surface geometry.) **Bump mapping:** Simpler technique, only encode hight difference from mesh. **Lighting and shadow:** can be baked into texture to save computational costs.
Conversion: $n = (2r-1, 2g-1, 2b-1)^T$, usually normalized.
Texture coordinates: Define a mapping from surface coordinates (points on triangle) to points in texture domain. Surface-to-texture space mapping is provided as per vertex attributes.

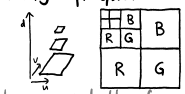
Filtering textures: Solves the problem of aliasing due to undersampling a high-resolution texture.
Minification: Area of screen pixel maps to large region of texture → filtering required. Happens when one texel corresponds to far less than a pixel on screen e.g. object is very far away.
Magnification: Area of screen pixel maps to tiny region of the texture → interpolation needed. One texel maps to many screen pixels, e.g. object very close to camera.

Mipmap: Prefiltering texture data to remove high frequencies
Computing d: mipmap $d = \log_2 L$

$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$

Tri-linear filtering have a continuous d:

↳ can do overblurring in one direction → anisotropic filtering



always get the four values where point lies in between.

Benefits: Small storage overhead (33%), for each operation: constant filtering cost and number of texels accessed.

Texture lookup sampling operation: For each covered screen sample (x,y): 1) compute u,v (via evaluation of attribute equasions) 2) compute du/dx, du/dy, dv/dx, dv/dy from adjacent screen samples. 3) compute d. 4) convert normalized (u,v) to (texel_u, texel_v) 5) Compute required texels in window of filter 6) Load required texels (tri-lin→8) 7) perform tri-linear interpolation

Graphics Pipeline

Depth buffer: For each coverage sample point, the depth buffer (z-buffer) stores depth of closest triangle at this sample point that has been processed by the renderer so far. Grayscale value of sample point: white = far, black = close
Occlusion using z-buffer: For a new point check if its depth < z-buffer value, if true, update z-buffer, and set color buffer value, else it is occluded.
can handle interpenetrating surfaces, works with super sampling. Range limited, that's why we have z-near, z-far
Z-fighting: common visual artifact that occurs when two planes are closely aligned Enable: glEnable(GL_DEPTH_TEST), Disable: glDisable(...)

Opacity: described as alpha value: fully opaque: $\alpha = 1$, fully transparent: $\alpha = 0$, maps linearly
Over operator: Composite image B with opacity α_B over

image A with opacity α_A. Composit color: $C = \alpha_B B + (1-\alpha_B)\alpha_A A$ and composite opacity: $\alpha_C = \alpha_B + (1-\alpha_B)\alpha_A$. Not commutative!
Premultiplied alpha: $A' = [\alpha_A A_R \ \alpha_A A_G \ \alpha_A A_B \ \alpha_A]^T$, $B' = [\alpha_B B_R \ \alpha_B B_G \ \alpha_B B_B \ \alpha_B]^T$ and $C' = B' + (1-\alpha_B)A'$ to save computations
Color buffer update: If passing z-test do not update the z-buffer and render triangles from back to front.
Mixture of opaque and transparent: render opaque surfaces as usual, then disable z-buffer update and render semi-transparent surfaces in back-to-front order. If depth test passed, triange is composited over contents of color buffer.

End-to-end rasterization pipeline (OpenGL, Direct3D)

Inputs: Vertex and primitive data, param. to compute vertices in normalized coordinates (e.g. transform matrices), param. for colors of fragments (texturies)

Operations on Vertices — Vertex processing: Transform triangle vertices into camera space. Apply perspective projection transform to transform triangle vertices into normalized coordinate system

Vertex Stream

Operations on primitives (lines, triangles...) — Primitive processing: Discard triangles that lie completely outside the unit cube (culling), clip triangles that go beyond to the cube (clipping).

Primitive Stream

Operations on fragments — Fragment generation: Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h). Triangle preprocessing: compute edge equations and triangle attribute equations

Fragment Stream

Operations on fragments — Fragment processing: Sample coverage, evaluate attributes z,u,v at all covered samples. Compute triangle color at sample point (interpolation, sample texture map...)

Shaded fragment stream

Operations on screen samples — Screen sample operations: Perform depth test (if enabled) and update z-buffer. Update color buffer (if depth test passed)

Input/Output: uniforms: (v/f) global constants for every vertex e.g. light position, texture map. uniform vec3 lightPos

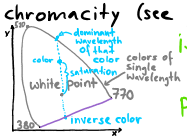
Attributes: (v) vertex-specific e.g. vertex position, normal. in vec4 position; Varyings: (v, f) values passed from vertex to fragment shader. Interpolated across primitive e.g. fragment color out vec3 color_out; gl_Position: (v) proj_mat * modelview_mat * position (position translated to camera coords, model→world→camera)
gl_FragColor: (f) is a vec4 that needs to be set to color.
Data types: float, int, bool, uint, vec{2,3,4}, ivec{2,3,4}, bvec{2,3,4}, mat{2,3,4} sampler{1,2,3}D (for textures). ↳access by . : xyzw, rgba or stpq.
Functions: void name(in ..., out ..., inout ...); can do overloading, no recursion.
Built-in: radiens, degrees, sin, cos, tan, asin, acos, atan, pow, exp, log, exp2, log2, sqrt, abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, length, distance, dot, cross, normalize, outerProduct, transpose, reflect
Textures: gl_FragColor = texture2D(texWood (sampler2D), texCoords (vec2))
Skybox: Box of 6 2D textures, in a single cubemap textureCube(tex, texDir) if viewer at center→correct distortion (sky is at infinity)

Rendering Equation

Color models:

Additive color model: used for combining colored lights e.g. RGB. (like emission, all together give white) $\begin{pmatrix}C\\M\\Y\end{pmatrix} = \begin{pmatrix}1\\1\\1\end{pmatrix} - \begin{pmatrix}R\\G\\B\end{pmatrix}$
Subtractive color model: used for combining paint colors e.g. CMYK. (like absorbtion subtracted from white, all together black)
HSV: hue, saturation, value Dimensions correspond to natural notions of "characteristics" of color. Used for color picking, similar to HSL
Y'CbCr: Y': perceived luminance (intensity), Cb: blue-yellow deviation from gray. Cr: red-cyan deviation from gray. (useful for compression)
YIQ: Y as luma information (grayscale), I for in-phase (orange-blue) Q for quadrature (purple-green). Advantages for natural & skin colors. based on psycho physical properties of the human eye.
RGB: Useful for monitors and displays.

CIE Experiment: subject is shown two stimuli at the same time, one with the pure spectral color, the other a linear combination of the three primaries (RGB). Subject can control how much primaries were dimmed and asked to match the second stimulus to the first. → find how humans perceive color. Can also add red light to reference if impossible to match →negative red values.
xyY colorspace: Y controls luminance, x and y control chromacity (see plot) match: $\delta(\lambda - v) \triangleq \bar{r}(v)r(\lambda) + \bar{g}(v)g(\lambda) + \bar{b}(v)b(\lambda)$



isolines: a line with about constant distance the border.
purple line: consists of non-spectral colors at the border.
XYZ-colorspace: XYZ to xyz: $x = \frac{X}{X+Y+Z}$ $Y = \frac{Y}{X+Y+Z}$
xyY to XYZ: $X = x\frac{Y}{y}$ $Z = \frac{Y}{y} - x\frac{Y}{y} - Y$

Measurements of light/color:

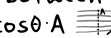
Radiometry: systems of units and techniques for measuring electromagnetic radiation (rubberball analogy)
Geometric model of light: Photons travel in straight lines, represented by rays. (good approximation if $\lambda \ll$ size)

Radient Energy	Radient Energy Density
(total number of hits)	(hits per unit area)
Radient Flux	**Irradiance**
(total hits per second)	(hits per second per unit area)

Energy carried by a photon: $Q = \frac{h \cdot c}{\lambda}$ λ: 390-700nm (color)
Flux: energy per unit time (Watts) received by the sensor or time-integral of flux is total radient energy: $Q = \int_{t_0}^{t_1} \Phi(t)dt$.
Luminous flux: Perceived power of light (Lumen) $F = c \cdot \int_{380nm}^{780nm} P(\lambda)V(\lambda)d\lambda$ 683 lm/W
where P(λ): relative spectral power density, V(λ): relative spectral sensitivity

Luminous intensity: $I^{[candela]} = dF/d\omega$, $F = \int I d\omega$
Luminance: $Y^{[\frac{candela}{m^2}]} = d^2F/dA\cos\theta d\omega$, $I = \int Y dA$, $F = \iint Y dA d\omega$
Illumination: $B^{[lux]} = dF/dA$

Lambert's law: Irradience at surface is proportional to cosine of angle between light direction and surface normal: $E = \Phi/\cos\theta \cdot A$



N-dot-L lighting: most basic way to shade a surface: dot product between unit surface normal and unit direction of light
directional lighting: same L for all. Isotropic point source.
Solid angle: $\Omega = A/r^2$, sphere has 4π steradians.
Differential solid angle: $dA = r^2 \sin\theta d\theta d\phi$, $d\omega = \sin\theta d\theta d\phi$
Radiance: fundamental quantity that characterizes the distribution of light in the environment
irradience→ $E = \int_{H^2} L(\omega)\cos\theta \, d\omega$ $L(\omega)$: radience in direction ω, θ: angle between ω and normal

Rendering equation:

$$L_o(p, \omega_i) = L_e(p, \omega_o) + \int_{H^2} f_r(p, \omega_i \to \omega_o) L_i(p, \omega_i) \cos\theta_i \, d\omega_i$$

L_o: outgoing/observed radiance, p: point of interest, ω_i: direction of interest, L_e: emitted radience (e.g. light source), ω_o: all directions in hemisphere, L_i: incoming radiance, $\cos\theta_i$: angle between incoming direction and normal (Lambert's law), f_r: scattering function

BRDF: Bidirectional Reflectance Distribution Function encodes behavior of light that bounces off surface, given incoming direction ω_i, how much gets scattered in outgoing direction ω_o: distribution $f_r(\omega_i \to \omega_o) \geq 0$, $\int_{H^2} f_r(\omega_i \to \omega_o)\cos\theta d\omega_i \leq 1$, $f_r(\omega_i \to \omega_o) = f_r(\omega_o \to \omega_i)$ "Helmholtz reciprocity" (tiny little mirrors)

Reflection: process by which light incident on a surface interacts with the surface such that it leaves on the same incident side without change in frequency.
Basic reflection functions: ideal specular  , ideal diffuse  , glossy specular  , retro-reflective 
Models of scattering: bounce off surface , bounce around inside surface  , transmitted through surface  , absorbed and re-emmitted: 

Phong Reflection Model: Ambient: light that shines independent of viewpoint and angle of the light ray hits the obj.

Diffuse: General direction of the light, which is reflected regardless of the viewer's position. Specular: The shiny light reflection. $I = \underbrace{I_a k_a}_{\text{ambient}} + \underbrace{I_p(k_d(N \cdot L)}_{\text{diffuse}} + \underbrace{k_s(R \cdot V)^n)}_{\text{specular}}$
Light param. material param.
Geometry param.

Raytracing

Rasterization vs raycasting

Rasterization: Proceed in triangle order, most processing based on 2D primitives (3D that was projected. Store depth buffer.

Raytracing: Proceeds in screen sample order, never have to store depth buffer (just current ray), natural order for rendering transparent surfaces. Must store entire scene.
Shadow mapping: Render scene (depth buffer only) from location of light. Everything "seen" (depth test success) from this PoV is directly lit, if depth test fail → in shadow
Shadows: ray tracing: shoot "shadow" rays towards light source from points where camera rays intersect scene. If nothing in the way → directly lit, else → in shadow.
Environment mapping: approximate appearence of reflective surface by placing a ray origin at location of reflective object, render six views (for a cube). Use camera ray reflected about surface normal to determine which texel in cube is "hit".
Reflections: raytracing: recursive ray tracing, compute a secondary ray from surface in reflection direction.

Geometric queries

Distance between two points: (a_1, a_2) and (p_1, p_2) → $|(a_1-p_1, a_2-p_2)|$
Closest point on a 2D line: given 2D line $N^Tx = c$ (N: unit normal), plug in: $N^T(p+tN) = c$ and compute $p+tN = p + (c - N^Tp)N$
Closest point line segment: find closest point on line, check if between endpoints ($a + t(b-a)$, check if $t \in [0,1]$) else closest endpoint
Point-line intersection: plug point in line equation.
Line-line intersection: $ax = b$, $cx = d$ ⇒ $\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$
Intersecting ray with implicit surface: all points s.t. $f(x)=0$ and ray: $r(t) = o + td$ ⇒ solve $f(r(t)) = 0$ for t.
Ray-plane intersection: Given plane $N^Tx = c$, ray $r(t) = o + td$ replace x with ray equation, solve for t. ⇒ point = $o + \frac{c - N^To}{N^Td} d$
Ray-triangle intersection: Parameterize triangle given by vertices p_0, p_1, p_2 (barycent. coords): $f(u,v) = (1-u-v)p_0 + up_1 + p_2$, solve for u, v, t: $\begin{bmatrix} p_1-p_0 & p_2-p_0 & -d \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix}^T = o - p_0$
Transforming plane normal: Given a plane P with normal n. P is transformed to P' using transformation matrix M. This means that $\forall p \in P. \exists! p' \in P'.\ p' = Mp$
Prove that $n' = (M^{-1})^Tn$ is normal to P': Using the Hesse normal form we know that $p^Tn = c$ for all $p \in P$ Thus for all $p' \in P'$ we have $p'^Tn' = (Mp)^T((M^{-1})^Tn) = p^TM^T(M^{-1})^Tn = p^T(M^{-1}M)^Tn = p^Tn = c$.

Accelerating geometric queries

Bounding volume hierarchy (BVH): Tree structure with interior nodes represent subset of primitives in scene, store aggregate bounding box of all primitives in subtree and leaf nodes containing a list of primitives.
↳ good BVH structure partition with equal number of primitives, minimize overlapping of children, avoid empty spaces.
Cost of making a partition: $C = C_{trav} + p_A C_A + p_B C_B$.
C_{trav}: cost of traversing interior node, p: probability ray intersects the bounding box of child node $C_{A,B}$: costs of intersection (recursive)
↳ approximation: $C = C_{trav} + \frac{S_A}{S_N}|A| C_{isect} + \frac{S_B}{S_N}|B| \cdot C_{isect}$ (surface area heuristic)
Primitive vs space partitioning: partition primitives into disjoint sets (may overlap spatially) vs splitting space into disjoint regions (primitives may overlap)
Uniform grid: Partition space into voxels (e.g. # primitives) each containing the primitives that overlap it. Walk through volume in order.
K-D trees: recursively partition space via axis aligned planes interior nodes ≙ spatial splits. Traversal front-to-back, stop at first hit (make sure intersection is within current leaf node.) adaptive! also quad/octree (are easy to build)

Rigging, FK and IK

Animation rigs: user-defined mapping between a small number of parameters and the deformation of a high-res mesh
Blend shapes: simplest type of rig. Simplest scheme: Take linear combination of vertex positions. Spline used to control choice of weights over time.

Cage-based deformers: Embed model in a coarse mesh (cage). Deform mesh → reconstruct high-res geometry.
↳ too restrictive, hard to have direct control

Skeletal animation: animate just the skeleton, then have the mesh "follow" automatically. (<< DoF)
Full simulation: Simulate muscles, fatty tissues, skin, interaction with environment,.. "right" answer, but very demanding

Forward Kinematics: Hierarchy of affine transformations, joints: local coordinate frames, bones: vectors between consecutive pairs of joints. Each non-root bone defined in frame of unique parent. Changes to parent frame affect all descendent bones.

Inverse Kinematics: Given goal(s) for end effector, compute the joint angles. **Basic idea:** write down distance between final point and target, set up objective. Compute gradient w.r.t. angles, apply gradient descent.

The Uncanny Valley

Motion capture: provides sparse signals, from these full body motion reconstructed. Often-times posed as an optimization problem (e.g. inverse kinematics)

Optimization Problem Standard form: $\min_{x \in \mathbb{R}^n} f_0(x)$, $f_i(x) \le b_i$, $i = 1, \dots, m$
Not always feasible, not always unique. **Unconstraint optimality conditions:** $\nabla f_0(x^*) = \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_n \end{bmatrix}(x^*) = 0$, $\nabla^2 f_0(x^*) \ge 0$ $\nabla^2 f_0 = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$ (Hessian)
$\succeq$: positive semidefinite $u^TAu \ge 0$ for all u.
Taylor series: $f(x) \approx f(x_0) + \langle \nabla f(x_0), x - x_0 \rangle + \langle \nabla^2 f(x_0)(x - x_0), x - x_0 \rangle / 2$.

Gradient descent: $x_{k+1} = x_k + \tau f_0'(x_k)$, make τ very small (1D), for (nD): $x_{k+1} = x_k - \tau \nabla f_0(x_k)$. **Newton's method:** $x_{k+1} = x_k - \tau (\nabla^2 f_0(x_k))^{-1} \nabla f_0(x_k)$ works well for convex problems.

Physically-Based Animation

The animation equation: $F = ma$ or $\ddot{q} = F/m$, often in generalized coordinates: $q = (x_0, x_1, \dots x_n)$, $\dot{q} = (\dot{x}_0, \dot{x}_1, \dots, \dot{x}_n)$
Ordinary differential equation: $\frac{d}{dt} q = f(q, \dot{q}, t)$ f: velocity func.
Higher order → first order: dummy variables e.g. $\dot{q} = F/m$ ⇒ $\begin{matrix} \dot{q} = v \\ \dot{v} = F/m \end{matrix}$ $\frac{d}{dt}\begin{bmatrix} q \\ v \end{bmatrix} = \begin{bmatrix} v \\ F/m \end{bmatrix}$

Rigid body: a solid body in which deformation is zero or so small it can be neglected. i.e. the distance between any given two points remains constant over time.
Linear velocity: $v(t) = \dot{x}(t)$
Angular velocity: $\omega(t)$ encoded with base at $x(t)$. magnitude → angular velocity, direction → spin axis.
Quaternion: $q(t)$, $\dot{q}(t) = \frac{1}{2}\omega(t) q(t)$
Force: $F = ma = m\dot{v} = m\ddot{x}$ where m is total mass.
Torque: τ (force applied not at $x(t)$) $\tau = (r(t) - x(t)) \times F$
Linear momentum: $p(t) = m\,v(t)$, $F(t) = \dot{p}(t)$
Angular momentum: $L(t) = I(t)\omega(t)$, $\tau(t) = \dot{L}(t)$
Center of mass: $x(t) = 1/M \cdot \sum_i m_i r_i$ for discrete mass points m_i at r_i and a total mass of M.

Rigid-Body dynamics: The kinematic state of a rigid body can be described by the position $x(t)$ of the center of mass, the linear momentum $L(t)$. Write down the ODEs that govern the dynamics of a rigid body are known at any time: $\dot{x}(t) = P(t)/m$, $\dot{P}(t) = F_i(t)$, $\dot{q}(t) = \frac{1}{2}(I^{-1}(t)L(t))\,q(t)$
$\dot{L}(t) = (r_i(t) - x(t)) \times F_i(t)$. For momentum/pos: $P(t+\delta t) = P(t) + \delta \cdot \dot{P}(t)$, $x(t+\delta t) = \delta t \dot{x}(t+\delta t)$
Handling collision event: Either adjust the state of the bodies after collisions by updating forces that act on bodies, or by updating momenta of involved bodies.

Lagrangian Mechanics: 1) Write down kinetic energy K, 2) Write down potential energy U, 3) Write down Lagrangian $\mathcal{L} = K - U$, 4) Dynamics given by Euler-Lagrange equation: $\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}} = \frac{\partial \mathcal{L}}{\partial q}$
Why useful? often easier to come up with (scalar) energies than forces. Very general, works in any kind of generalized coords.
Example: $K = \frac{1}{2} v^T m v$, $U = mg^T g$. $\frac{\partial \mathcal{L}}{\partial \dot{q}} = -mg$, $\frac{\partial \mathcal{L}}{\partial \dot{q}} = mv \Rightarrow \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}} = ma$
Example 2: $K = \frac{1}{2} m l^2 \dot{\theta}^2$, $U = -mgl\cos\theta$, $\frac{\partial \mathcal{L}}{\partial \dot{q}} = ml^2\dot{\theta}$, $\frac{\partial \mathcal{L}}{\partial q} = -mgl\sin\theta \Rightarrow \ddot{\theta} = -\frac{g}{l}\sin\theta$

Particle Systems: Model complex phenomena as large collections of particles (each have behavior described by (physical or non-physical) forces). Very common in computer graphics since easy to understand, simple equation for each particle and easy to scale up/down. (sometimes need many!)
Example: Mass-spring system: connect two particles x_1, x_2 by

spring of length L: $F_{spring} = -K\left(\frac{|x_1-x_2|}{L_0} - 1\right)\frac{x_1-x_2}{|x_1-x_2|}$ L_0: rest length, $|x_1-x_2|$: current length

Numerical Integration

Forward Euler: $q_{k+1} = q_k + \tau f(q_k)$ f: velocity at current time
↳ intuitive, but not very stable: e.g. $\dot{u} = -au$, $a>0 \rightsquigarrow u_n = (1-\tau a)^n u_0$

Backward Euler: $q_{k+1} = q_k + \tau f(q_{k+1})$ new config implicit → solve
↳ much harder, but unconditionally stable, "dampening" $\rightsquigarrow u_n = \left(\frac{1}{1+\tau a}\right)^n u_0$

Symplectic Euler: update velocity using current config, update config using new velocity. e.g. $v_{k+1} = v_k + \tau F_m/m$, $q_{k+1} = q_k + \tau v_{k+1}$
↳ easy to implement, no numerical dampening very often used.

Criteria: stability, accuracy, consistency, symmetry, computational efficiency

Partial Differential Equations

PDE: Implicitly describe function in terms of its time derivatives and its spatial derivatives in equation.
↳ solve for function of time and space: u(t, x)

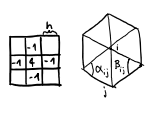
Linear: e.g. $\dot{u} = au''$ (diffusion equation) non-linear: e.g. $\dot{u} + uu' = au''$ (Burgers' equation) hig her order: e.g. $\ddot{u} = au''$ (wave equation) has order (2,2)

Model equations: "elliptic": Laplace equation: $\Delta u = 0$
"Parabolic": heat equation: $\dot{u} = \Delta u$, "hyperbolic": wave equation: $\ddot{u} = \Delta u$

Numerical solving of PDEs: pick PDE that models phenomena of interest. Pick spatial discretization: **Lagrangian:** particle based, track and read their measurements, **Eulerian:** grid-based record at fixed locations in space, pick time discretization: approximate derivatives in time: forward, backward, symplectic Euler, have an update rule and repeatedly solve to generate animation

Laplace operator: Nabla operator: $\nabla = \left(\frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_d}\right)$, Laplace operator: $\Delta = \nabla \cdot \nabla = \sum_{k=1}^{d} \frac{\partial^2}{\partial x_k^2}$ e.g. $\Delta u = \nabla \cdot \nabla u = \frac{\partial u^2}{\partial x_1^2} + \dots + \frac{\partial u^2}{\partial x_n^2}$
→ takes function, gives second derivative.

Discretizing Laplacian: two common ways:
Grid: $(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1})/h^2$
Triangle mesh: $\frac{1}{2}\sum_j (\cot\alpha_{ij} + \cot\beta_{ij})(u_j - u_i)$



Boundry conditions

Dirichlet: boundry data always set to fixed values e.g. $\Phi(0) = a$
Neumann: specify derivative (difference) accross boundry $\Phi'(0) = u$
Robin: mixed boundry conditions. e.g. $\Phi'(0) + \Phi(0) = p$

Numerically solving PDEs

Laplace Equation: want to solve $\Delta u = 0$, plug in one of the discretizations e.g. grid: $\frac{a-b-c-d-e}{h} \rightsquigarrow \frac{4a-b-c-d-e}{h} = 0 \Leftrightarrow a = \frac{1}{4}(b+c+d+e)$
At the solution, each value is average of neighbors. Idea: keep averaging the neighbors: Jacobi method. correct, but slow convergence
1D-Laplacian with Dirichlet: a line between boundries, always works
1D-Laplacian with Neumann: a line, only works for same boundry conditions
2D with Dirichlet: always works, 2D with Neumann works if net flux $\overset{\text{(at boundry)}}{=} 0$

Heat equation: want to solve $\dot{u} = \Delta u$. Use e.g. forward Euler $u_{i,j}^{k+1} = u^k + \frac{\tau}{h^2}(4u_{i,j}^k - u_{i+1,j}^k - u_{i-1,j}^k - u_{i,j+1}^k - u_{i,j-1}^k)$. (easy to implement)

Wave equation: want to solve $\ddot{u} = \Delta u$. Convert to first order (in time) equations $\dot{u} = v$, $\dot{v} = \Delta u$. Evaluate spatial derivative $\frac{u^{k+1} - 2u^k + u^{k-1}}{\tau^2} = \Delta u^k$. Integrate forward in time e.g. with symplectic Euler

Animation

Applications: entertainment, education, communicatio useful in robotics, computer vision engineering and scientific visualization.

Principles of animation: Squash and stretch: defining the rigidity & mass of an object by distorting its shape during motion
Timing: spacing actions to define the weight & size of objects & the personality of characters. Anticipation: preparation of an action
Staging: presenting an idea so that it is unmistakenly clear.
Follow through & overlapping action, straight ahead action & pose-to-pose action, slow in and out, arcs: the visual path of action for natural movement, exaggeration, secondary action, appeal, solid drawing

Generating motion: artist-directed (e.g. keyframing), data-driven (e.g. motion capture), procedural (e.g. simulation)
Keyframing: Specify important events only, computer fills in the rest via interpolation/approximation. Events can be position, color, light intensity, camera configuration.
Motion capture: Track points, compute joint angles using inverse kinematics and use these angles for animation. (1 day MoCap, 3 day clean up)

Splines: piecewise polynomial function that interpolates data
Natural splines: cubic polynomials, 1st & 2nd derivatives match and $p_0''(t_1) = 0 = p_{m-1}''(t_n)$. Interpolates, C^2-continuous, not local.
Hermite/Bézier splines: cubic polynomials, interpolate + 1st derivative is given tangent. Interpolates, not C^2-continuous, local
B-Splines: not interpolating, C^2-continuous, local
Spline desiderata: Interpolation: spline passes exactly through data points, continuity in C^2, locality: moving one point does not affect whole curve. ⇒ impossible to have all at once.

Shape parameterization for deformation: $x(u,v) = \sum_k p_k B_k(u,v)$
p_k: control points, $B_k(u,v)$: basis function that yields the mixing or blending ratio.
Blending weights: Linear basis functions (e.g. bi-linear basis function $B_{(0,1,2,3)} = [uv, (1-u)v, u(1-v), (1-u)(1-v)]$) Cubic basis functions (e.g. Bezier surface)

Skeletons: Hierarchical Deformation, a N-joint skeleton is given by: Root frame w.r.t. world R_0, relative joint coordinate frames $R_1, R_2, \dots, R_N$ with $R_j = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Skinning: Moving the mesh with the bones
Rigid skinning: Assign each vertex to one bone/joint j
pos. of vert. in animated mesh: $\hat{v}_i = F_j(A_j)^{-1} v_i$ ← pos. of vert. in reference mesh
joint j in animated mesh → alternative: $v = \omega R_j (\omega R_j)^{-1} v'$ → joint j in reference mesh
Only works for vertices close to bones or small rotations.
Linear blend skinning: Assign each vertex to multiple bones/joints: $\hat{v}_i = \sum_{j=1}^{N} w_{ij} F_j (A_j)^{-1} v_i$ where w_{ij}: influence of joint j on vertex i
How to choose weights? Often manually, automatically (still research)
Candy-wrapper effect: How to fix: better weights, introduce auxiliary joints/bones, employ better interpolation schemes for transformations, pose space deformers (PSD)

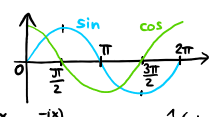
Character animation: traditionally based on motion clips and blend trees (special kind of animation state machine for smooth transitions). Problem: scripted motion quickly becomes repetitive.

Responsive and richer character motion: simulate the virtual world as the real world: A simple controller based on an inverted pendulum model: PD control with desired motion (fast & simple model to find joint angles), Virtual forces for velocity tuning (moving the center of mass through torques)
Inverted pendulum for balance (given desired velocity, outputs next foot placement for balance)
Statistics: record a lot of the real world: try to map desired direction to next state, body trajectory as desired direction for supervised learning (they used a phase-dependent cyclic network

Pseudo physics: Idea using a kinematic controller (e.g. blend tree): detect collision event, launch a passive simulation, then blend between kinematic controller and simulated ragdoll over time.
↳ popular in games since simulation is costly and hard to control.

Ingredients for responsive character in AR: 3D reconstruction of environment, motion model with kinematic locomotion (active movement), terrain adaption with inverse kinematics, pseudo-physics pertubations (e.g. short duration ragdoll)

	0/0	30°/π/6	45°/π/4	60°/π/3	90°/π/2
sin α	0	1/2	√2/2	√3/2	1
cos α	1	√3/2	√2/2	1/2	0



$e^{ix} = \cos x + i \sin x$ $\sin x = \frac{1}{2i}(e^{ix} - e^{-ix})$ $\cos x = \frac{1}{2}(e^{ix} + e^{-ix})$
$\cos\varphi = \frac{a \cdot b}{|a| \cdot |b|}$ $\sin \rightsquigarrow \frac{G}{H}$, $\cos \rightsquigarrow \frac{A}{H}$, $\tan \rightsquigarrow \frac{G}{A}$