Visual Computing Summary

Lukas Wolf

ETH Zurich
HS 2020

Contents

Part I
Visual Computing

1 Images, Segmentation and Filtering

1.1 The Digital Image

1.1.1 What is an image?

A signal is a function depending on some variable with physical meaning. An image is a continuous function in 2 variables, or 3 variables (+ time, then it's a video). Brightness is usually the value of the function (can be other physical values such as temperature, pressure, depth, etc.).

1.1.2 The Digital Camera

Is a charge coupled device (CCD). It contains a lense and a 2D sensor array. Once in a while the photons of each array of "photosites" are emptied into "buckets" of electrical charge. They contain charge proportional to the incident light intensity during exposure. The analog to digital conversion (ADC) measures the charge and digitizes the result.
The "blooming" effect (very bright lines in pictures) happens because of saturation of a photosite (and therefore the charge goes into neighboring buckets).
"Bleeding" or smearing occurs because during transit the buckets still accumulate some charges. Therefore this effect is worse for short shutter times (only problem with electronic shutter).
Dark current occurs because of thermally-generated charge in CCDs. They give non-zero output even in darkness. Dark current may also occur because of other reasons.

1.1.3 CMOS Cameras

The sensor elements are the same as in CCD, each photo sensor does have its own amplifier (leads to more noise, reduces by subtracting 'black' image).
While CMOS is cheap, needs low power, has random pixel access and smart pixels, CCD has a higher fill rate. CMOS videos sensors do have the rolling shutter issue, the sequential read-out of lines may lead to round lines if they are moving during the read-out.

1.1.4 Sampling, Reconstruction, Quantization

If we undersample a signal, trivially, information is lost. The surprising result is that the signal gets indistinguishable from lower frequency signals. **Aliasing** means signals "traveling in disguise" as other frequencies, i.e. when sampled can't be distinguished anymore (too less samples).

The **Nyquist Frequency** (a.k.a. Nyquist-Shannon sampling theorem) is half the sampling frequency of a discrete signal processing system. The signal's max frequency (bandwidth) must be smaller than this.
There exist different sampling grids, like cartesian, hexagonal or non-uniform.
Quantizing real valued functions will give us digital values. After quantization, the original signal cannot be reconstructed anymore. This is in contrast to sampling, as a sampled but not quantized signal can be reconstructed. Simple quantization uses equally spaced levels with k intervals, $k = 2^b$. Usual intervals are 8 bit greyscale, or 8 bit per color channel for RGB.

1.1.5 Image Properties

Some image properties are **image resolution** (how many pixels in total), **geometric resolution** (how many pixels per area) and **radiometric resolution** (how many bits per pixel).
A common model for **image noise** is additive gaussian noise. Others are poisson (shot noise) or rician (appears in MRI) noise.
The **signal to noise ratio (SNR) s** is an index of image quality:
$s = \frac{F}{\sigma}$, where $F = \frac{1}{XY} \sum_{x=1}^{X} \sum_{y=1}^{Y} f(x, y)$. One also often uses the Peak Signal to Noise Ratio (PSNR) instead, where one uses F_{max}.

1.1.6 Concepts of Color Cameras

The **Prism color camera** spearates light in 3 beams using a dichroic prism. It thus requires 3 sensors and precise alignment, one wins good color separation. High separation, hight framerate, 3 sensors.
In a **Filter mosaic camera** one has a filter directly on the sensor and uses alternating RGB cells. Average separation, low cost, high framerate, but aliasing.
In a **Filter wheel** camera, one rotates multiple filters in front of lens and allows more than 3 color bands. Good separation, average cost, low framerate, motion doesn't work.

1.2 Image Segmentation

1.2.1 What is Image Segmentation?

Segmentation is the ultimate classification problem. Once solved, Computer Vision is solved. Image Segmentation partitions an image into regions of interest.
A complete segmentation of an image I is a finite set of regions $R_1, ..., R_N$, such that $I = \bigcup_{i=1}^{N} R_i$ and $R_i \cap R_j = \emptyset, \forall i \neq j$.
Segmentation algorithms must be chosen and evaluated with an application in mind.

1.2.2 Thresholding

Thresholding is a simple segmentation process. It produces a binary image B and labels each pixel in or out of the region of interest by comparison of the

greylevel with a threshold T: $B(x,y) = \begin{cases} 1, \text{ if } I(x,y) \geq T \\ 0, \text{ if } I(x,y) < T \end{cases}$
We choose T by trial and error or comparing it with ground truth. We may also choose some other distance, e.g. the **Mahalanobis distance** (measure of the distance between a point and a distribution). There are also some automatic methods (such as ROC curves).
One can segment images much better by eye than through a thresholding process, because one can improve results by considering image context (surface coherence).

1.2.3 Chromakeying

"Plain" distance measure is used to distinguish relevant pixels: $I_\alpha = |I - g| > T$.
Problems: Variation is not the same in all 3 channels.

1.2.4 ROC Analysis

An **ROC (Receiver operating characteristic)** curve characterizes the performance of a binary classifier, namely the tradeoff of the TP fraction (sensitivity) $\frac{TP}{P}$ against the FP fraction (1-specificity) $\frac{FP}{N}$.
An ROC curve always passes through (0,0) and (1,1). A perfect system would have a line from (0,0) to (0,1) to (1,1). A random classifier has a straight line from (0,0) to (1,1).
As operating point, we choose the point on the ROC curve with gradient $\beta = \frac{N}{P}\frac{V_{TN}+C_{FP}}{V_{TP}+C_{FN}}$, with values and costs assigned, respectively. For simplicity, we often set $V_{TN} = V_{TP} = 0$, therefore only assign costs for wrongly classified points.
In real-life, we use two or even three separate sets of test data: A training set, for tuning the algorithm; a validation set, for tuning the performance score; and an unseen test set to get a final performance score on the tuned algorithm.

1.2.5 Connected Regions

We need to define which pixels are neighbors. We can define a **4-neighborhood** (a cross around the pixel in the center), or an 8-neighborhood (a full 3x3 square). These are then called 4- and 8-connected path between pixels $p_1, ..., p_i$, respectively.
A region is **4-connected**, if it contains a 4-connected path between any two of its pixels. Same holds for 8.

Connected components labelling labels each connected component of a binary image with a separate number. **Foreground labelling** only extract the connected components of the foreground.

Region growing starts from a seed point or region and adds neighboring pixels that satisfy the criteria defining a region. One repeats until we can include no

more pixels. **Seed selection** can be done manually or automatically, e.g. from a conservative thresholding. One can also use multiple seeds.

The **Greylevel distribution model** uses mean and standard deviation in the seed region and includes if $(I(x,y)-\mu)^2 < (n\sigma)^2$, with e.g. $n=3$. We can update the mean and standard deviation after every iteration.

Snakes or active contours are polygons, i.e. an ordered set of points joined up by lines. Each point on the contour moves away from the seed while its image neighborhood satisfies an inclusion criterion. Often the contour has smoothness constraints. The algorithm iteratively minimized an energy function: $E = E_{tension} + E_{stiffness} + E_{image}$

1.2.6 Foreground-Background Segmentation

For **Distance Measures** one can use a plain background subtraction metric such as: $I_\alpha = |I - I_{bg}| > T$. where I_{bg}=background image.
When possible, fit a Gaussian model per pixel, with mean I_μ and standard deviation I_Σ.
Even better is $I_\alpha = \sqrt{(I-I_{bg})^T \Sigma^{-1} (I-I_{bg})} > T$ (T=4 for example), which is the beforementioned Mahlanobis Distance with $I_\mu = I_{bg}$. Σ is here the background pixel appearance covariance matrix (computed separately for each pixel, from many examples; sometimes need more than one Gaussian, therefore use GMMs).

1.2.7 Adding spatial relations with Markov Random Fields / Graph Cuts

One uses a MRF with a pixel label for each node (pixel), and each node is connected via an edge to all its neighboring pixels. One now wants to minimize the Energy:

$$E(y;\theta,data) = \sum_i \psi_1(y_i;\theta,data) + \sum_{i,j\in edges} \psi_2(y_i,y_j,data).$$

This can be solved via graph cuts, by adding a source (label 0) and a sink (label 1) to the graph (the connected pixels). Many of these energy minimization problems can be approximated by solving a maximum flow problem in a graph (and thus, by the max-flow min-cut theorem, define a minimal cut of the graph). The Cut is separating source and sink and meanwhile minimizing the energy (collection of edges, i.e. the connection from foreground to background).

Iterated Graph Cut may use k-means for learning colour distributions and then use graph cuts to infer the segmentation. It gets difficult with camouflage and low contrast or also fine structured images.

1.3 Spatial Domain: Morphological Operations

1.3.1 What are Morphological Operators?

Morphological operators are local pixel transformations for processing region shapes. Most often they are used on binary images. The logical transformations are based on comparison of pixel neighborhoods with a pattern.

1.3.2 Stucturing Elements and Set Notation

Morphological operations take two arguments: a **binary image** and a **structuring element**. One compares the structuring element to the neighborhood of each pixel. This determines the output of the morphological operation. The structuring element is also a binary array (e.g. $\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$) with an origin (e.g. center pixel).
We can think of the binary image and the structuring argument as sets containing the pixels with value 1.
We will use the same **set notation** that we are used from other courses with intersection $\cap$, union $\cup$, complement I^C and the empty set $\emptyset$.

1.3.3 Fitting, Hitting and Missing

$$\text{S fits I at x, if } \{y : y = x + s, s \in S\} \subset I.$$
$$\text{S hits I at x, if } \{y : y = x - s, s \in S\} \cap I = \emptyset.$$
$$\text{S misses I at x, if } \{y : y = x + s, s \in S\} \cap I = \emptyset.$$

The structuring element is said to fit the image if, for each of its pixels set to 1, the corresponding image pixel is also 1. Similarly, a structuring element is said to hit, or intersect, an image if, at least for one of its pixels set to 1 the corresponding image pixel is also 1.

1.3.4 Erosion and Dilation

The image $E = I \ominus S$ is the erosion of image I by structuring element S.
$E(x) = \begin{cases} 1\text{,if S fits I at x} \\ 0\text{,if otherwise} \end{cases}$, $E = \{x : x + s \in I \text{ for every } s \in S\}$.
The image $D = I \oplus S$ is the dilation of image I by structuring element S.
$D(x) = \begin{cases} 1\text{, if S hits I at x} \\ 0\text{, if otherwise} \end{cases}$, $D = \{x : x - s \in I \text{ for every } s \in S\}$.

1.3.5 Opening and Closing

The opening of I by S is $I \circ S = (I \ominus S) \oplus S$, Open = Dilation(Erosion(I))
The closing of I by S is $I \bullet S = (I \oplus S) \ominus S$, Close = Erosion(Dilation(I)).

1.3.6 Examples

Eight-neighbor erode
A.k.a Minkowsky subtraction: Erase any foreground pixel that has one eigth-connected neighbor that is background.

Eight-neighbor dilate
A.k.a Minkowsky addition: Paint any background pixel that has one eight-connected neighbor that is foreground. One uses it because of the need of smooth region boundaries for shape analysis. Remove noise and artefacts from an imperfect segmentation.

1.3.7 Morphological Filtering

To remove holes in the foreground and islands in the background, do both opening and closing. The size and shape of the structuring element determine which features survive. In the absence of knowledge about the shape of features to remove, use a circular structuring element.

Granulometry provides a size distribution of distinct regions or "granules" in the image. We open the image with increasing structuring element size and count the number of regions after each operation. Creates a **"morphological sieve"**.

Hit-and-miss transform searches for an exact match of the structuring element. $H = I \otimes S$ is the hit-and-miss transform of image I by structuring element S. Simple form of template matching, an example would be e.g. a upper-right corner detector.

1.3.8 Thinning and Thickening

are defined in terms of the hit-and-miss transform:
The thinning of I by S is $I \oslash S = I \setminus (I \otimes S)$,
The thickening of I by S is $I \odot S = I \cup (I \otimes S)$,
Dual operations: $(I \odot S)^C = I^C \oslash S$.
These operations are often performed in sequence with a selection of structuring elements $S_1, S_2, ..., S_n$. Several sequences of structuring elements are useful in practice. These are usually the set of rotations of a single structuring element, sometimes called the **Golay alphabet**.

1.3.9 Skeletonization and the Medial Axis Transform

The skeleton and medial axis transform (MAT) are stick-figure representations of a region $X \subset \mathbb{R}^2$. One starts a grassfire at the boundary of the region. The skeleton is the set of points at which two fire fronts meet (e.g. for a circle it would be the center).
An alternative skeleton definition would be that the skeleton is the union of

centres of maximal discs within X, where a **maximal disc** is a circular subset of X that touches the boundary in at least two points. The MAT is the skeleton with the maximal disc radius retained at each point.
Skeletonization can be done using morphology. E.g. use the structuring element $\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. The n-th skeleton subset is $S_n(X) = (X \ominus_n B) \setminus [(X \ominus_n B) \circ B]$, where $\ominus_n$ denotes n successive erosions. The skeleton is the union of all the skeleton subsets: $S(X) = \bigcup_{n=1}^{\infty} S_n(X)$

1.3.10 Reconstruction from Skeletonization

One can **reconstruct** region X from its skeleton subsets: $X = \bigcup_{n=0}^{\infty} S_n(X) \oplus_n B$. We can reconstruct X from the MAT but **not** from S(X).

Applications and Problems
The skeleton/MAT provides a stick figure representing the region shape which is used in object recognition, in particular, character recognition. Problems that may arise are e.g. the definition of a maximal disc, which is poorly defined on a digital grid. It is also sensitive to noise on the boundary. Sequential thinning output is sometimes preferred to skeleton/MAT.

1.4 Convolution and Filtering

1.4.1 Image Filtering and Kernels

Image filtering is modifying the pixels in an image based on some function of a local neighborhood of the pixels. In **linear shift-invariant Filtering**, linear means a linear combination of the neighbors, and shift-invariant means doing the same for each pixel. This is useful to perform low-level image processing operations, smoothing and noise reduction, sharpen images, and detect or enhance features.
Filters can be seen as taking a dot-product between the image and some vector. Filtering the image is a set of dot-products. Filters look like the effects they are intended to find.

Linear operations can be written as:

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

where I is the input image, I' is the output of the operation, and k is the **kernel** of the operation. $N(m,n)$ is a neighborhood of (m,n). Operations are called **shift-invariant**, if k does not depend on (x,y): using the same weights everywhere. "Flipping the kernel" means transposing the matrix.

1.4.2 Correlation and Convolution

The linear operation of correlation (e.g. template matching) is defined as:

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

One can represent the linear weights as an image K.
The linear and associative operation of convolution (e.g. point spread function) is defined as:

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

Correlation and Convolution are conceptually very different operations, but if $K(i,j) = K(-i,-j)$ (i.e. symmetric kernels), then both operations are equal.

1.4.3 Separable Kernels and the Gaussian Kernel

Separable filters (kernels) can be written as $K(m,n) = f(m) \cdot g(n)$. For a rectangular neighborhood we have $I(m,n) = f \cdot (g \cdot I(N(m,n)))$.
Smoothing kernels (low-pass filters) are of the form $\frac{1}{10}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$.

Gaussian Kernel
Idea: weight contributions of neighboring pixels by nearness. It is a symmetric and separable filter, i.e. the response is identical in any direction. It is of the form: $G_\sigma = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$. The constant factor at front makes the volume sum up (integral) to 1.
Smoothing with a Gaussian results in just a blurred version, whereas smoothing with a box filter has a convolution effect. We will see strong edges in the picture, therefore a box filter is not a good lowpass blurring filter.
The amount of smoothing with a Gaussian Kernel depends on σ and the window size (width $> 3\sigma$). Repeated convolution by a Gaussian filter produces the **scale space** of an image.

Gaussian Smoothing Kernel Perks
The Gaussian smoothing kernel is rotationally symmetric, has a single lobe (neighbor's influence decreases monotonically) and still one lobe in frequency domain (no corruption from high frequencies). It has a simple relationship to σ and is easy to implement efficiently.

1.4.4 Differentiation and Convolution

Recall for a 2D function f(x,y) we can approximate the differentiation $\frac{\delta f}{\delta x} \approx \frac{f(x_{n+1},y)-f(x_n,y)}{\Delta x}$. This is linear and shift-invariant, so must be the result of a convolution. In particular, it's using $\begin{pmatrix} -1 & 1 \end{pmatrix}$. But since this leads to a very noisy kernel, we rather use Prewitt or Sobel kernel (more low-pass), to get a more averaged edge detection.

1.4.5 Differential Filters

The **Prewitt operator** is used in image processing, particularly within edge detection algorithms. Technically, it is a discrete differentiation operator, com-

puting an approximation of the gradient of the image intensity function. An example is: $\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$. Another example is the **Sobel operator**, which has -2 and 2 instead of -1 and 1 on the middle row. If something at one side is brighter than on the other, these operators will highlight this difference. The response of both filter is 0 (averaging out).

High-pass filters: E.g. the **Laplacian operator** will highlight edges, but much stronger respond to a local dot.

1.4.6 Image Sharpening

Image sharpening is also known as Enhancement and increases the high frequency components to enhance edges. $I' = I + \alpha|k * I|$, where k is a high-pass filter kernel and α a scalar in $[0, 1]$.

1.4.7 Integral images

Integral images (also known as summed-area tables) allow to efficiently compute the convolution with a constant rectangle. One uses a recursion formula to sum up in a single pass.

1.5 Image Features

1.5.1 Template Matching

Problem: locate an object, described by a template $t(x, y)$ in the image $s(x, y)$. We want to find something that looks roughly like our template (e.g. an eye). We can search for the best match by minimizing the mean squared error between the template and any region in the image, i.e. the error

$$E(p, q) = \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} [s(x, y) - t(x - p, y - q)]^2$$

Equivalently, we can maximize the area correlation. The are correlation is equivalent to convolution of image $s(x, y)$ with the impulse response $t(-x, -y)$. Note that we have to remove the mean before template matching to avoid bias towards bright image areas.

1.5.2 Edge detection

Idea (continuous-space): Detect the local gradient (square root of sum of squares of partial differentials). We can use the difference $\begin{pmatrix} -1 & 1 \end{pmatrix}$, the central difference $\begin{pmatrix} -1 & [0] & 1 \end{pmatrix}$, or the Prewitt or Sobel operators that we have already seen. Another possible operator is the Roberts operator, that looks like $\begin{pmatrix} [0] & 1 \\ -1 & 0 \end{pmatrix}$ or also $\begin{pmatrix} [1] & 0 \\ 0 & -1 \end{pmatrix}$, where the squared number is the center.

1.5.3 Laplacian Operator

The Laplacian operator detects discontinuities by considering the second derivative:

$$\bigtriangledown^2 f(x,y) = \frac{\delta^2 f(x,y)}{\delta x^2} + \frac{\delta^2 f(x,y)}{\delta y^2}.$$

The operator is isotropic. An isotropic operator in an image processing context is one which applies equally well in all directions in an image, with no particular sensitivity or bias towards one particular set of directions. Discrete-space approximation can be done by a convolution with 3×3 impulse response: $\begin{pmatrix} 0 & 1 & 0 \\ 1 & [-4] & 1 \\ 0 & 1 & 0 \end{pmatrix}$, where -4 can also be -8. The second derivative will go through zero if the edge profile (gradient) is steep, whereas the first derivative only grows in absolute value and decreases again after the gradient decreases. Therefore the zero crossing mark edge location when using the Laplacian operator.

The Laplacian operator is sensitive to very fine detail and noise (blur image first). It responds equally to strong and weak edges (suppresses "edges" with low gradient magnitude).
Blurring of an image with Gaussian and Laplacian operator can be combined into convoluition with **Laplacian of Gaussian (LoG)** operator.

1.5.4 Canny Edge Detector

Combine noise reduction and edge enhancement.

1. Smooth image with a Gaussian filter
2. Compute gradient magnitude and angle (Sobel, Prewitt,...)

$$M(x,y) = \sqrt{(\frac{\delta f}{\delta x})^2 + (\frac{\delta f}{\delta y})^2} \text{ and } \alpha(x,y) = tan^{-1}(\frac{\delta f/\delta y}{\delta f/\delta x})$$

3. Apply non-maximum suppression to gradient magnitude image to get rid of spurious response of edges
4. Double thresholding to detect strong and weak edge pixels
5. Reject weak edge pixels not connected with strong edge pixels through other weak edges.

The **Canny non-maximum suppresion** quantizes each edge to one of four directions (horizontal, -45 deg, vertical, +45 deg), i.e. quantizing $\alpha(x,y)$ to one of four different states. Gradient direction is always perpendicular to edges. If $M(x,y)$ is smaller than either of its neighbors in direction normal/perpendicular to the edge, then suppress it. Otherwise keep it.

For the **Canny thresholding and suppression of weak edges** we use double-thresholding of gradient magnitude, i.e.

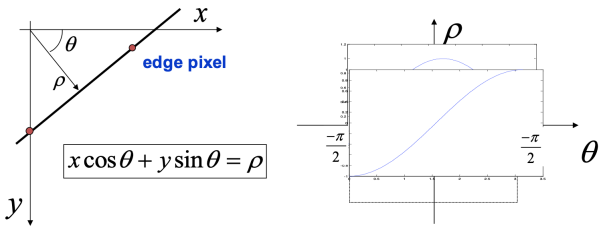
Strong edge: $M(x,y) \geq \theta_{high}$
Weak edge: $\theta_{high} > M(x,y) \geq \theta_{low}$.

In a typical setting we have $\frac{\theta_{high}}{\theta_{low}} = 2$ or 3. We use region labeling of edge pixels and reject regions without strong edge pixels. A weak edge point is not connected to a strong edge point.

1.5.5 Hough Transform

Problem: fit a straight line (or curve) to a set of edge pixels. The **Hough Transform** is a generalized template matching technique. Consider the detection of straight lines $y = mx + c$.

- Subdivice (m, c) plane into discrete "bins", initialize all bin counts by 0
- Draw a line in the parameter space m, c for each edge pixel x, y and increment bin counts along line
- Detect peak(s) in (m, c) plane
- Alternative parameterization avoids infinite-slope problem



- For **Circle detection** of undetermined radius, use 3-d Hough transform for parameters (x_0, y_0, r)

1.5.6 Detecting corner points

Many applications benefit from features localized in (x, y). Since edges are well localized in only one direction, we can use them to detect corners. Desirable properties of a corner detector are accurate localization, invariance against shift, rotation, scale and brightness change; and robustness against noise as well as high repeatability.

1.5.7 Local Displacement sensitivity

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

which can be approximated with:

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

where

$$\mathbf{M} = \begin{pmatrix} f_x^2(x,y) & f_x(x,y)f_y(x,y) \\ f_x(x,y)f_y(x,y) & f_y^2(x,y) \end{pmatrix}$$

is a rank-1 matrix. $f_x(x,y)$ and $f_y(x,y)$ are the horizontal and vertical image gradients, respectively. Iso-sensitivity curves are ellipses.

1.5.8 Feature Point Extraction

$$SSD \approx \Delta^T M \Delta$$

Find points for which the following is large, i.e. maximize the eigenvalues of M:

$$\min SSD \approx \Delta^T M \Delta, \text{ for} ||\Delta|| = 1$$

1.5.9 Keypoint detection (Harris Corner Detector)

Often based on eigenvalues λ_1, λ_2 of the *structure matrix aka normalmatrix aka second-moment matrix*. Measure the *cornerness* with:

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

For better localization of corners one can give more importance to central pixels by using a Gaussian weighting function:

$$M' = \textstyle\sum_{(x,y)\in \text{window}} G(x - x_0, y - y_0, \sigma) \cdot M$$

One can compute the subpixel localization by fitting a parabola to the cornerness function.
The Harris corner detector is invariant to brightness offset, shift rotation but **not** invariant to scaling.

1.5.10 Lowe's SIFT Features

The scale-invariant feature transform (SIFT) recovers features with position, orientation and scale.

Position

- Look for strong responses of **DoG filter** (Difference-of-Gaussian) and only consider local maxima:

$$DOG(x,y) = \tfrac{1}{k} e^{-\frac{x^2+y^2}{(k\sigma)^2}} - e^{-\frac{x^2+y^2}{\sigma^2}}, k = \sqrt{2}$$

Scale

- Look for strong responses of DoG filter over scale space
- Only consider local maxima in both position and scale

- Fit quadratic around maxima for subpixel accuracy

Orientation:

- Create a histogram of local gradient directions computed at selected scale
- Assign canonical orientation at peak of smoothed histogram
- Each key specifies stable 2D coordinates (x,y, scale, orientation)

SIFT Descriptor

- Thresholded image gradients are sampled over 16×16 array of locations in scale space
- Create array of orientation histograms
- 8 orientations x 4×4 historgram array = 128 dimensions

2 Transformations

2.1 Fourier Transform

The Fourier Transform is essentially a change of basis. We represent the function on a new basis. We apply a linear transformation to transform the basis (dot product with each basis element). In the following expression, u and v select the basis element, so a function of x and y becomes a function of u and v. Basis elements have the form $e^{-i2\pi(ux+vy)} = \cos 2\pi(ux+vy) - i \sin 2\pi(ux+vy)$.

$$F(g(x,y))(u,v) = \int\int_{\mathbb{R}^2} g(x,y) e^{-i2\pi(ux+vy)} dxdy$$
$$F = \mathbf{U}f$$

where f is the vectorized image and U the Fourier transform base (also ossible wavelets, steerable pyramids, etc.)

2.1.1 Fourier Basis Functions

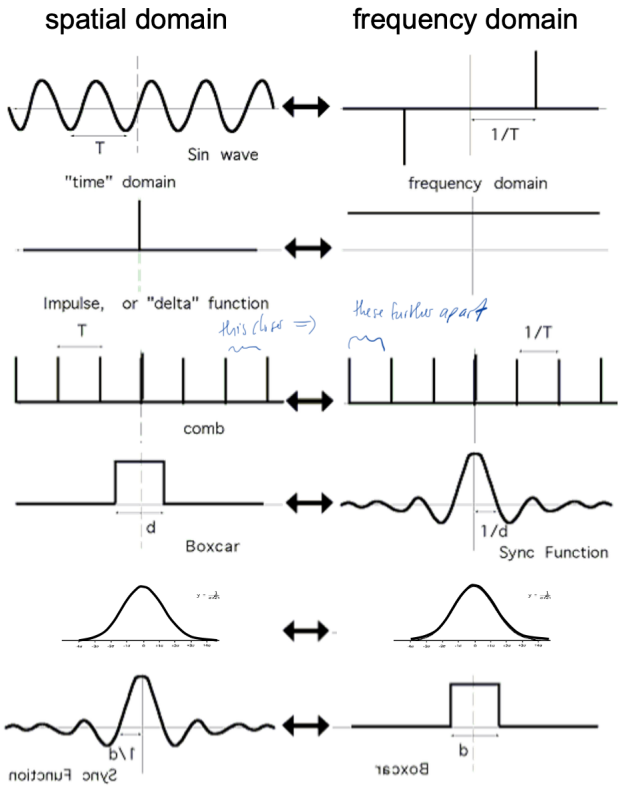
Basis functions of Fourier transform are eigenfunctions of linear systems. If we plot the vector (u,v) in the complex plane, then the magnitude (the further away from origin) of the vector gives the **frequency** and the direction gives **orientation**. In other words, for small u and v we need big x and y to get high frequency.

The Fourier transform of a real function is complex. Note that all natural images have about the same magnitude transform, hence, phase seems to matter but magnitude largely doesn't.

2.1.2 Facts about Fourier Transform

- The Fourier transform is linear
- There is an inverse FT $f = U^{-1}F$
- scale function down $\iff$ scale transforms up, i.e. if we would squeeze a Gaussian, the frequency domain Gaussian will get wider
- The FT of a Gaussian is a Gaussian

2.1.3 Fourier Transform of Important Functions



2.1.4 Convolution Theorem

The Fourier transform of the convolution of two functions is the product of their Fourier transforms. Therefore filtering (i.e. convolution) is just a simple product in the frequency domain.

$$F.G = \mathbb{U}(f * *g)$$

The Fourier transform of the product of two functions is the convolution of the Fourier transforms (cfr. sampling).

$$F * *G = U(f.g)$$

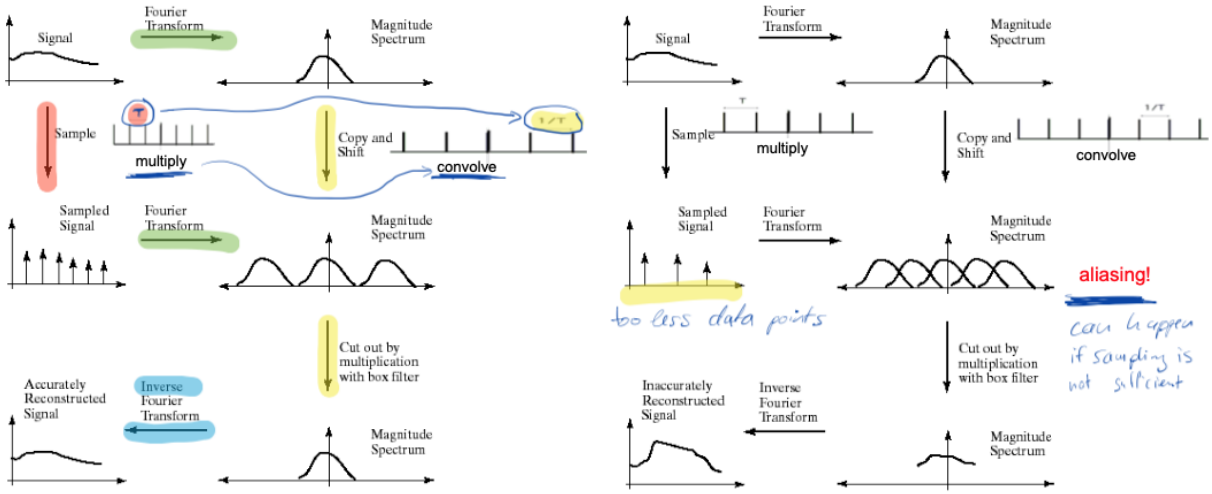
2.1.5 Sampling

Sampling means going from the continuous to the discrete world. We want to be able to approximate integrals sensibly. This leads to the delta function.

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

The **delta function** can be infinitely high with zero width, i.e. a constant area function. If it is $\frac{1}{h}$ wide, it will be h high.
The Fourier transform of a sampled signal will result in

$$F(\text{Sample}(f(x,y))) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} F(u-i, v-j).$$



Proper sampling requires low-pass filtering before sampling. The message of FT us that high frequencies can lead to problems with aliasing if not properly sampled.
The solution is suppressing high frequencies before sampling. For that we wither multiply the FT of the signal with something that suppresses high frequencies, or convolve with a low-pass filter. A filter whose FT is a box is bad, because the filter kernel has infinite support. A common solution is using a Gaussian, since multiplying FT by Gaussian is equivalent to convolving the image with a Gaussian.

Nyquist Sampling Theorem
The sampling frequency must be at least twice the highest frequency

$$\omega_s \geq 2\omega$$

If this is not the case, the signal needs to be bandlimited before sampling, e.g. with a low-pass filter.

Computation of 2D FT
2D FT can be computed as a sequence of 1D FTs. The **Fast Fourier Transform (FFT)** can compute Discrete FT very fast (by using symmetries).

2.1.6 Reconstruction

The basic pipeline is:

1. Low-pass filtering
2. Sampling
3. Reconstruction

As reconstruction filter, we can use square pixels, Gaussians, Bilinear interpolation (linear interpolation in both directions) or perfect reconstruction filters.

Motion blurring
Each light dot is transformed into a short line along the x_1-axis:

$$h(x_1, x_2) = \frac{1}{2l}[\theta(x_1 + l) - \theta(x_1 - l)]\delta(x_2)$$

.
Image Restoration Problem
The inverse kernel $\tilde{h}(x)$ should compensate the effect of the image degradation $h(x)$, i.e.

$$(\tilde{h} * h)(x) = \delta(x)$$

$\tilde{h}$ may be determined more easily in Fourier space:

$$F[\tilde{h}](u, v) \cdot F[h](u, v) = 1$$

To determine $F[h]$ we need to estimate (1) the distortion model h(x) (point spread function) or $F[h](u, v)$ (modulation transfer function), and (2) the parameters of $h(x)$, e.g. r for defocussing.
The Fourier transformation for the motion blur kernel results in

$$F[h](u, v) = \frac{\sin(2\pi ul)}{2\pi ul} := sinc(2\pi ul).$$

Therefore the inverse kernel is $\tilde{h}(u) = sinc(2\pi ul)$ and the Fourier transform of it is $F[\tilde{h}](u) = 1/\tilde{h}(u)$.
Problems that arise are:

- Convolution with the kernel h completely cancels the frequencies $\frac{\nu}{2l}, \nu \in \mathbb{Z}$
- Noise amplification for $F[h](u, v) \ll 1$.

Avoiding Noise Amplification We can use a regularized reconstruction filter:

$$\tilde{F}[\tilde{h}](u,v) = \frac{F[h]}{|F[h]|^2+\epsilon}$$

Singularities are avoided by the regularization ϵ. The size of ϵ implicitly determines an estimate of the noise level in the image, since we discard signals which are dampened below the size ϵ.

2.2 Unitary Transforms

2.2.1 Basics of Unitary Transforms

A digital image can be written as a matrix (for colored images three matrices). The matrix can be transformed to a vector $\vec{f}$, which makes the math much easier.

Linear Image Processing algorithms can be written as

$$\vec{g} = H\vec{f}$$

Almost all image processing systems contain at least some linear operators. We want to choose H such that $\vec{g}$ separates the salient features from the rest of the image signal and becomes sparse.

For **Unitary transforms** A holds:

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

If A is real values, i.e. $A = A^*$, the transform is **orthonormal**.

We sort the samples f(x,y) of an MxN image (or a rectangular block in the image) into column vector of length MN and compute the coefficients

$$\vec{c} = A\vec{f}$$

where A is a matrix of size $MN \times MN$. For any unitary transforms the **energy conservation** holds, i.e.

$$\|\vec{c}\| = \|\vec{f}\|.$$

Every unitary transform is simply a rotation of the coordinate system (and, possibly, sign flips). Vector lengths ("energies") are therefore conserved.

We can build an image collection F from n images $f_1, ..., f_n$ as $F = [f_1, ..., f_n]$. The image collection **auto-correlation function** is given as

$$R_{ff} = E[f_i \cdot f_i^H] = \frac{F \cdot F^H}{n}.$$

Energy is conserved, but often will be unevenly distributed among coefficients. The **autocorrelation matrix** is given as

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

Mean squared values ("average energies") of the coefficients c_i are on the diagonal of R_{cc}:

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

The **eigenmatrix** Φ of a autocorrelation matrix R_{ff} is a unitary matrix with columns that form a set of eigenvectors of R_{ff}, i.e.

$$R_{ff}\Phi = \Phi\Lambda,$$

where Λ is a diagonal matrix of eigenvalues λ_i. R_{ff} is symmetric non-negative definite, hence $\lambda_i \geq 0$. R_{ff} is a normal matrix, i.e. $R_{ff}^{H}R_{ff} = R_{ff}R_{ff}^{H}$, hence a unitary eigenmatrix exists.

2.2.2 Karhunen-Loeve Transform (aka PCA)

The PCA (or diskrete Karhunen-Loeve transform) is a unitary transform with matrix $A = \Phi^{H}$ where the columns of Φ are ordered according to decreasing eigenvalues. It is a non-parametric method of extracting relevant information from data. The transform coefficients are pairwise uncorrelated:

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

No other unitary transform packs as much energy into the first j coefficients (j arbitrary). The mean squared approximation error by choosing only first j coefficients is minimized.

2.2.3 Basis Images and Eigenimages

For any unitary transform, the inverse transform

$$\vec{f} = A^{H}\vec{c}$$

can be interpreted in terms of the superposition of **basis images** (columns of A^{H}) of size MN. If the transform is a KL transform, the basis images, which are the eigenvectors of the autocorrelation matrix R_{ff}, are called **eigenimages**, which is therefore a special case of basis image.
If energy concentration works well, only a limited number of eigenimages is needed to approximate a set of images with small error. These eigenimages form an optimal linear subspace of dimensionality J.

Eigenimages for recognition
To recognize complex patterns (e.g. faces), large portions of an image (say of size MN) might have to be considered. High dimensionality of "image space" means high computational burden for many recognition techniques (e.g. nearest-neighbor search requires pairwise comparison with every image in a data base). Note that the images I_i alls need to be aligned, e.g. the tip of the nose should be at the same pixel. The transform $\vec{c} = W\vec{f}$ can reduce dimensionality from

MN to j by representing the image by j coefficients.
The idea is to tailor a KL-transform to the specific set of images of the recognition task to preserve the salient features.
In a simple recognition task we just measure the euclidian distance between images and the best match wins:

$$\arg\min_i D_i = \|I_i - I\|.$$

This is computationally quite expensive, since it requires the presented image to be correlated with every image in the database.
Using **eigenspace matching** (PCA) we use the closest rank-k approximation property of SVD to compute a much cheaper comparison of the two images. In order to do this, we transform both the presented image and the compared images into a k-dimensional subspace and compare them there.

2.2.4 Eigenfaces

An eigenface is the additive construction of the average face plus a linear combination of eigenfaces. It can be used for face recognition by nearest neighbor search in 8-d "face space". It can also be used to generate faces by adjusting 8 coefficients.
Compared to SSD matching, eigenspace matching will typically work better, because only the main characteristics are preserved and irrelevant details are discarded. The limitations of eigenfaces are that the differences due to varying illumination in different pictures of the same person can be much larger than differences between faces of different people.

2.2.5 Fisherfaces - Linear Discriminant Analysis (LDA)

Key ideas:

- Find directions where ratio of between to within individual variance is maximized.
- Linearly project to basis where dimension with good signal to noise ratio are maximized
- In contrast to the Eigenimage method (which maximizes "scatter" within the linear subspace over the entire image set - regardless of classification task) Fisher LDA maximized between-class scatter, while minimizing within-class scatter.

Solution: Generalized eigenvectors $\vec{w_i}$ corresponding to the K largest eigenvalues, i.e.

$$R_B \vec{w_i} = \lambda_i R_W w_i, i = \vec{1}, 2, ..., K$$

The problem is that the within-class scatter matrix R_w is at most of rank $L - c$, hence usually singular. We can apply a KLT/PCA transform to first reduce the

dimension of feature space to L-c (or less) and then proceed with Fisher LDA in low-dimensional space.
In the example of projecting samples for 2-dimensional classes onto a 1-dimensional subspace using KLT or FLD, PCA preserves maximum energy, but the two classes are no longer distinguishable. FLD separates the classes by choosing a better 1d subspace. Fisherfaces especially solve the varying illumination problem better than KLT/PCA.

2.2.6 JPEG

Usually one doesn't resolve high frequencies too well, therefore we can use this fact to compress images. JPEG is based on a block-based **Discrete Cosine Transform (DCT)**. It is first divided into 8×8 blocks, then quantized by a matrix Q, splitted into DC and AC parts, and finally is encoded as DC and AC Huffman code blocks. The decoding routine is just the reversed process. DCT is a variant of discrete Fourier transform (with real numbers) which can be implemented very fast.
Small blocks are faster and correlation exists only between neighboring pixels, large blocks lead to better compression in smooth regions. DCT enables image compression by concentrating most image information in the low frequencies. The decoder computers the inverse DCT (IDCT).

Huffman code is a type of entropy coing. The more probable a code occurs, the shorter its representation. The average code word length with this method is $H = -\sum p(s) \log_2 p(s)$, which is optimal.

JPEG2000 is a discrete wavelet transform (DWT) based compression standard. JPEG 2000 decomposes the image into a multiple resolution representation in the course of its compression process. This pyramid representation can be put to use for other image presentation purposes beyond compression.

2.3 Pyramids and Wavelets

2.3.1 Scale-space representations

From an original signal f(x) generate a parametric family of signals $f^t(x)$, where fine-scale information is successively suppressed. The family of signals can be generated by successive smoothing with a Gaussian filter.

Applications of scaled representations are:

- Search for correspondence: look at coarse scales, then refine with finer scales
- Edge tracking: a "good" edge at fine scale has parents at a coarser scale
- Control of detail and computational cost in matching: e.g. finding stripes - terribly important in texture representation

2.3.2 The Gaussian Pyramid

- Smooth with Gaussians because gaussian * gaussian = another gaussian
- Synthesis: smooth and sample
- Analysis: take the top image
- Gaussians are low pass filters, so representation is redundant

2.3.3 The Laplacian Pyramid

- Synthesis: preserve difference between unsampled Gaussian pyramid level and Gaussian pyramid level. The different levels act as band pass filter, therefore represent spatial frequencies (largely) underrepresented at other levels.
- Analysis: reconstruct Gaussian pyramid, take top layer.

2.3.4 Oriented Pyramids

The Laplacian pyramid is orientation independent. Apply an oriented filter to determine orientations at each layer. By clever design, we can simplify synthesis. This represent image information at a particular scale and orientation.

2.3.5 Haar Transform

In mathematics, the Haar wavelet is a sequence of rescaled "square-shaped" functions which together form a wavelet family or basis. Wavelet analysis is similar to Fourier analysis in that it allows a target function over an interval to be represented in terms of an orthonormal basis.
We sample $h_k(x)$ at m/N for $m = 0, ..., N-1$. Transition at each scale p is localized according to q. Basis images of 2-D (separable) Haar transform are outer products of two basis vectors.
The Haar transform contains two major sub-operations: Scaling captures information at different frequencies, translation captures information at different locations. It does have a relatively poor energy compaction though.

2.3.6 Lifting

The lifting scheme is a technique for both designing wavelets and performing the discrete wavelet transform (DWT). In an implementation, it is often worthwhile to merge these steps and design the wavelet filters while performing the wavelet transform. This is then called the second-generation wavelet transform.
We use L "lifting steps" (analysis filters), and can perfectly reconstruct (biorthogonality is directly built into the lifting structure) with the synthesis filters.

3 Optical Flow

Optical flow describes the movement of pixels. Motion can also be the only cue for segmentation in a picture/video. Even impoverished motion data can elicit a strong percept (e.g. a motion capture system, i.e. a person with LEDs on clothes in front of a camera that tracks the movements).
Optical flow is defined as the apparent motion of brightness patterns (or colors). Ideally, the optical flow is the projection of the three-dimensional velocity vectors on the image. This definition requires caution. Let's look at two examples that are ambiguous:

Uniform, rotating sphere **buf** Optical Flow = 0, or
No motion, but changing lighting **but** Optical Flow $\neq 0$.

3.1 Mathematical formulation: Brightness Constancy

Let $I(x, y, t)$ = brightness at (x, y) at time t. Then the **brightness constancy assumption** (or color constancy for RBG) is described by

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

It describes a small change of δt from one frame to the next. The assumption means that the pixel/object will look the same in the next frame. By using a taylor expension we end up with

$$\frac{dI}{dt} = \frac{\delta I}{\delta x}\frac{dx}{dt} + \frac{\delta I}{\delta y}\frac{dy}{dt} + \frac{\delta I}{\delta t} = 0 \; I_x u + I_y v + I_t \approx 0,$$

where e.g. I_x is the image gradient along x direction that can be computed e.g. with a Sobel filter. I_t is the temporal partial derivative, i.e. the difference between two frames. Since we have two unknowns in one equation, we need more constraints to solve for it. E.g. we can assume that the flow for all pixels is constant within a path (i.e. flow is locally smooth). For a 5×5 patch, we end up with 25 equations and can now solve for the two values u and v (e.g. by least squares $A^T Ax = A^T b$, which is what is done by Lucas-Kanade algorithm).

3.2 The Aperture problem

The aperture problem can be demonstrated by looking at a moving image through a small hole – the aperture. Different directions of motion can appear identical when viewed through an aperture. Let's define the horizontal speed as $u = \frac{dx}{dt}$ and the vertical speed by $v = \frac{dy}{dt}$. Then, for the partial derivatives I_x, I_y, I_t we have

$$I_x u + I_y v + I_t = 0,$$

i.e. the change of brightness due to motion should be compensated through time. This resembles just the optical flow constraint equation. The above equation defines a line in the (u,v) space. Any solution on the line will be fine.

3.3 What is optical flow?

It is an estimate of the observed projected motion field. It is not always well defined! Compare the following:

- Motion Field (or Scene Flow): projection of 3-D motion field
- Normal flow: observed tangent motion
- Optic flow: apparent motion of the brightness pattern (hopefully equal to motion field).

Consider the Barber pole illusion. It rotates around its axis but it looks like the lines are going up.

3.4 Regularization

The Horn–Schunck method of estimating optical flow is a global method which introduces a global constraint of smoothness to solve the aperture problem. The Horn-Schunck algorithm assumes smoothness in the flow over the whole image. Thus, it tries to minimize distortions in flow and prefers solutions which show more smoothness. The additional smoothness constraint is

$$e_s = \int\int ((u_x^2 + u_y^2) + (v_x^2 + v_y^2))dxdy$$

besides the OF constraint equation term

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

We now want to minimized $e_s + \lambda e_c$, which can be done by solving the **Euler-Lagrange equations**, which leads us to

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

where Δ is the Laplacian operator. These are coupled PDEs and can be solves using iterative methods and finite differences. More than two frames allow a better estimation of I_t. Information spreads from corner-type patterns.
The Horn-Schunck method makes errors at boundaries and is an example of regularisation (selection principle for the solution of ill-posed problems)

3.5 Lucas-Kanade

In computer vision, the Lucas–Kanade method is a widely used differential method for optical flow estimation developed by Bruce D. Lucas and Takeo Kanade. It assumes that the flow is essentially constant in a local neighbourhood of the pixel under consideration, and solves the basic optical flow equations

for all the pixels in that neighbourhood, by the least squares criterion. By combining information from several nearby pixels (i.e. a patch), the Lucas–Kanade method can often resolve the inherent ambiguity of the optical flow equation. It is also less sensitive to image noise than point-wise methods. On the other hand, since it is a purely local method, it cannot provide flow information in the interior of uniform regions of the image.
Assume a single velocity for all pixels within an image patch

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

and solve this with

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

On the LHS: sum of the 2×2 outer product tensor of the gradient vector

$$(\sum \Delta I \Delta I^T)\vec{U} = -\sum \Delta I I_t$$

Let $M = \sum (\Delta I)(\Delta I)^T$ and $b = \binom{-\sum I_x I_t}{-\sum I_y I_t}$. Then the algorithm computes $\vec{U}$ at each pixel by solving $MU = b$. M is singular, if all gradient vectors point in the same direction (e.g. along an edge). I.e. only normal flow is available (aperture problem).

Iterative Refinement: We estimate the velocity at each pixel during one iteration of Lucas-Kanade estimation. We then warp one image toward the other using the estimated flow field, refine the estimate and repeat the process.

3.5.1 Lucas-Kanade Coarse-to-fine

One of the assumptions was that the velocity is very small. Limits of the (local) gradient method are:

- Fails when intensity structure within window is poor
- Fails when the displacement is large (typical operating range is motion of 1 pixel per iteration)
- Also, brightness is not strictly constant in images (actually less problematic than it appears, since we can pre-filter images to make them look similar)

In Coarse-to-fine estimation we an image pyramid that has a different scale of u and v at each layer (e.g. 1.25, 2.5, 5, 10 pixels). We start at a coarse level and estimate some optical flow (will be very unprecise). We can then warp the images and apply Lucas-Kanade on the higher pyramid level. At the final level we reach the original resolution of the image. We have much less artifacts than without using pyramids.
One application of OF is slow motion(generate intermediate frames).

3.6 Parametric Motion Models

Parametric models derive the additional constraint required to solve the aperture problem by modeling the projection onto the image plane of surfaces moving in the 3-D space. Consequently, they rely on a segmentation of the frame into independently moving regions representing these surfaces. The motion of each region is described by a set of a few parameters, making it very compact in contrast to the non-parametric dense field description. These parameters are sufficient to synthesize or reconstruct the motion vector of any pixel in the image. Instead of every pixel having its own motion, we only store very few global parameters.
The energy that we want to minimize is

$$E(h) = \sum_{x \in \mathbb{R}} [I(x+h) - I_0(x)]^2,$$

where $h = \binom{\delta x}{\delta y}$ is the translation. We can now generalize to more complicated times of motion. E.g. The affine motion described by the matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

is able to describe any 2D transformation with only 6 parameters (together with the 2 from h). One can also use a planar perspective 3×3 matrix A.
We can further generalize to minimize the energy between I_0 and an arbitrary translation $I(f(x,y))$.

3.7 Residual Planar Parallax Motion

Epipolar geometry is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points.
The intersection of the two line constraints (epipolar line, brightness constancy constraint) uniquely defines the displacement.

3.8 Bayesian Optic Flow

Some low-level human motion illusions can be explained by adding an uncertainty model to Lucas-Kanade tracking.

3.9 SSD Tracking

For large displacements we can perform template matching as it was used in stereo disparity search.

1. Define a small area around a pixel as the template

2. Match the template against each pixel within a search area in next image
3. Use a match measure such as correlation, normalized correlation, or sum-of-squares difference
4. Choose the maximum (or minimum) as the match
5. Sub-pixel interpolation also possible

4 Video Compression

4.1 Perception of Motion

The human visual system is specifically sensitive to motion, our yes follow motion automatically. Some distortions are not as perceivable as in image coding (would be if we froze the frame). There is no good psycho-visual model available, though.
Visual perception is limited to < 24 Hz. A succession of images will therefore be perceived as continuous if the frequency is sufficiently high (cinema 24Hz, home tv 25/50 Hz). One still needs to avoid aliasing, that's why high-rendering frame-rates are desired in computer games (needed due to absence of motion blur).
Using a **interlaced video format** with a top field and a bottom field, we have 2 temporally shifted half images and can increase the frequency, e.g. from 25 to 50 Hz.

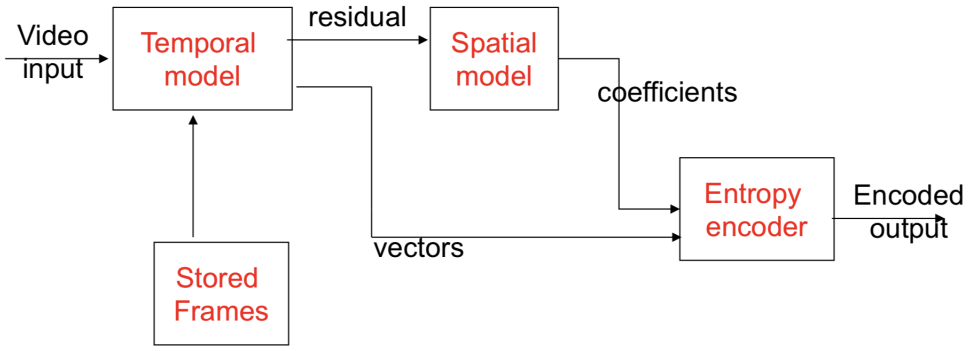
4.2 Lossy Video Compression

We can take advantage of redundancy, i.e. spatial correlation between neighboring pixels as well as temporal correlation between frames. We drop perceptually unimportant details.

4.2.1 Temporal Processing (Frame Differencing)

The simplest way would be to just use DCT or wavelets on all pictures of the video. This is very inefficient for nonuniform motion, i.e. real-world motion, and requires a large number of frame store (which leafs to delay).
Predictive methods gives us much better performance using only 2 frame stores. However, simple frame differencing is not enough. We want to predict the current frame based on previously coded frames. For that matter we define three types of coded frames:

- I-frame: Intra-code frame, coded independently of all other frames
- P-frame: Predictively coded frame, coded based on previously coded frame
- B-frame: Bi-directionally predicted frame, coded based on both previous and future coded frames



Temporal redundancy reduction may be ineffective when many scene changes occur (high motion).

4.2.2 Motion-compensated prediction

Simple frame differencing fails when there is much motion. MC prediction partitions the video into moving objects and describe the objects motion. This is in general very difficult. A practical approach is **Block-Matching Motion Estimation**:

- Partition each frame into blocks, e.g. 16×16 pixels
- Describe motion of each block, candidate blocks are all blocks in the frame. One can search for candidate blocks either as full search (examine all candidate blocks), or partial (fast) search: examine a carefully selected subset
- Use best matching blocks (e.g. by MSE) of reference frame as prediction of blocks in current frame, this gives us the motion vector
- Benefits: No object identification required, good and robust performance

The **motion vector** expresses the relative horizontal and vertical offsets (mv_1, mv_2), or motion, of a given block from one frame to another. Each block has its own motion vector. The collection of motion vectors for all the blocks in a frame gives us the **motion vector field**.
Motion is not limited to integer-pixel offsets. **Half-pixel motion vectors** reduce the prediction error through the averaging effect of spatial interpolation between the discrete pixel locations. For noisy sequences, the averaging effect reduces noise and leads the an improved compression. A practical Half-Pixel motion estimation algorithm would be:

1. Coarse step: perform integer motion estimation on blocks, find best integer-pixel MV

2. Fine step: refine estimate to find best half-pixel MV

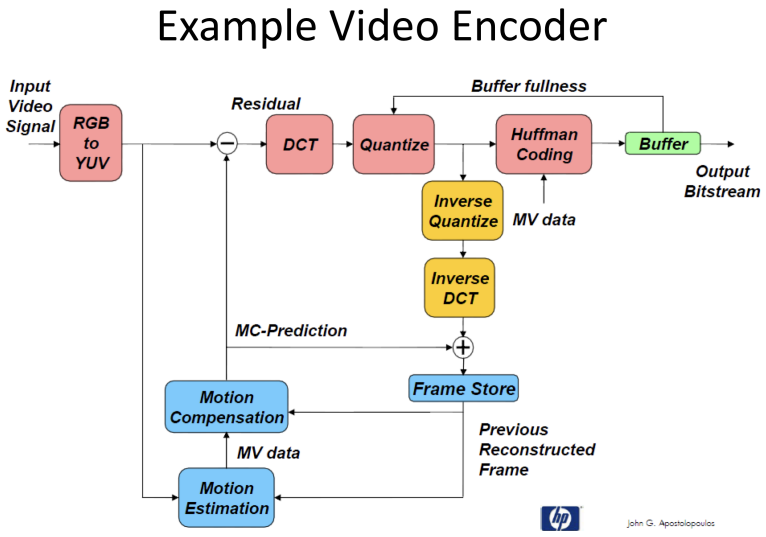
    - Spatially interpolate the selected region in reference frame
    - Compare current block to interpolated reference frame block
    - Choose the integer half-pixel offset that provides best match

Typically, bilinear interpolation is used for spatial interpolation. The resulting motion vector field is easy to represent (one MV per block) and useful for compressions, it assumes a translational motion model though, and breaks down for more complex motion. It also often produces blocking artifacts (OK for coding with Block DCT).

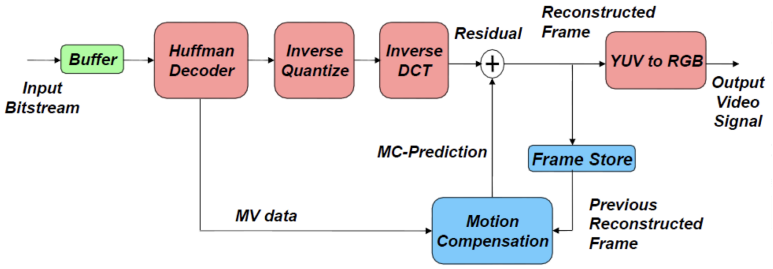
4.2.3 Basic Video Compression Architecture

We exploit the following redundancies:

- Temporal: MC-prediction (P and B frames)
- Spatial: Block DCT
- Color: Color space conversion



Example Video Decoder



Part II
Computer Graphics

5 Introduction to Computer Graphics

5.1 What is computer graphics?

Computer graphics is the use of computers to synthesize and manipulate visual information. In CG we want to build computational models of the real world. This requires sophisticated theory and systems:

- Theory: geometric representations, sampling, integration and optimization, perception, physics-based modeling, etc.
- Systems: parallel, heterogeneous processing, graphics-specific programming languages

5.2 How does a (digital) computer draw lines?

The display abstracts an image as a 2D grid of pixels. *Rasterization* means converting a continuous object to a discrete representation on a pixel grid. One method to color the discrete pixels is the *Diamond rule*: Imagine an upright diamond in each pixel and only color it if the line hits the diamond.
To rasterize along the line, we assume two endpoints and compute the slope. Then we can go along the line within a loop and increment pixels and color them.

6 Drawing Triangles

Triangles are very useful to build up other structures. When it comes to drawing triangles, we have to ask two questions:

- What pixels does the triangle overlap? ("coverage")
- What triangle is closest to the camera in each pixel? ("occlusion")

6.1 The visibility problem

In terms of rays, we have to know what scene geometry is hit by a ray far from the pixel through the camera's pinhole (coverage). Also we need knowledge about what object is the first hit along the ray (occlusion). One option to use the coverage of a pixel by a triangle is to color it according to the fraction that the triangle covers the pixel (e.g. 10 % coverage then 10 % red).

Computing the amount of overlap with analytical schemes can get quite tricky,

especially when considering interactions between multiple triangles. The alternative is to estimate the amount of coverage through sampling random dots in the pixel. Remember that sampling the function at a certain position is equivalent to multiplying it (inner product) by the dirac delta function at this position. Reconstruction is done by convoluting the sampled function with a reconstruction filter (e.g. box, triangle, gaussian, sinc, etc.).

We can think of coverage as a 2D signal like

$$coverage(x, y) = \begin{cases} 1, & \text{if the traingle contains point (x,y)} \\ 0, & \text{otherwise.} \end{cases}$$

Now what happens if a sample point lies exactly on the edge between two triangles? In *OpenGL/Direct3D* the sample is classified as within triangle if the edge is a "top edge" or "left edge". A top edge is a horizontal edge that is above all other edges, a left edge is an edge that is not exactly horizontal and is on the left side of the triangle. Note that a triangle can have one or two left edges.

6.2 Aliasing in displaying pixels

Remember that we can represent signals as a superposition of frequencies (FT). The individual frequencies are 2D sinusoids. Aliasing means that high frequencies in the original signal masquerade as low frequencies after reconstruction (due to undersampling). The signal can be perfectly reconstructed is sampled with period $T > \frac{1}{2\omega_0}$, where ω_0 is the maximal frequency in the signal. Note that for this perfect reconstruction we use a normalized *sinc* function (ideal reconstruction filter with infinite extent).
Unfortunately, signals are often not band-limited and, infinite extent of "ideal" reconstruction filter (sinc) is impractical for efficient implementations. To get better results we can increase the density of sampling, or even use supersampling (use multiple samples per pixel). We then have to *resample*, i.e. convert from one discrete samples representation to another (because a screen displays one sample value per screen pixel). This allows us to get more smooth objects, with pixels colored according to their supersampling statistics.

6.3 Sampling triangle coverage

We can check which points are in a triangle by computing the triangle edge equations from the projected positions of vertices. A sample point is then inside the triangle if it is "inside" all three edges.
Rather than testing all points on the screen, we can traverse them incrementally, only checking remaining points that are possibly in the triangle. There are many traversal orders possible, such as backtrack, zig-zag, Hilbert/Morton curves.

A modern approach is *tiled triangle traversal* We traverse the triangle in blocks

and test all samples in a block against the triangle in parallel. One advantage is that we can skip sample testing work if an entire block is early discovered as not being in the triangle. We can also perform wide parallel execution.

7 Transforms

7.1 Linear Maps

We limit our scope to *linear maps*, because all maps can be approximated by linear maps over a short distance (or small amount of time) by *taylor's theorem*. Linear maps are easy to solve and still very powerful.
A map is linear, if it maps vectors to vectors, and if for all vectors u, v and scalars a we have:

$$f(u+v) = f(u) + f(v), \text{ and}$$
$$f(au) = af(u).$$

Note that a linear map has the property $f(0x) = 0x = 0$.

7.2 Important Transforms

7.2.1 Scale

We differ between *uniform* scale $S_a(x) = ax$ and *non-uniform* scale $S_a b(x) = x_1 a e_1 + x_2 b e_2 = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} x$ (both in 2D). Uniform scale is a linear transform.

7.2.2 Rotation

$R_\theta =$ rotate counter-clockwise by θ. As angle changes, points move along *circular* trajectories. Shape does not change! Rotation is a rigid/isometric transform and preserves distances. Rotation is also a linear map. If we want to rotate about another point than the origin, move it to the origin, rotate, move back. Rotation can be written as $R_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} x$.

7.2.3 Reflection

$Re_y(x)$: reflection about y-axis. Reflections change *handedness*. To rotate around an arbitrary axis, perform all necessary rotations such that we can reflect about y-axis and then rotate all back.

7.2.4 Shear

$H_a(x) = x_1 \binom{1}{0} + x_2 \binom{a}{1} = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} x$. Here every point not on the x-axis is moved along the x axis by factor a.

7.2.5 Translation

$T_b(x) = x + b$. Translation moves all points along the line $\binom{b}{b}$. Note that translation is not linear, but *affine* (preserves lines and parallelism).

7.3 Matrix notation for linear transforms

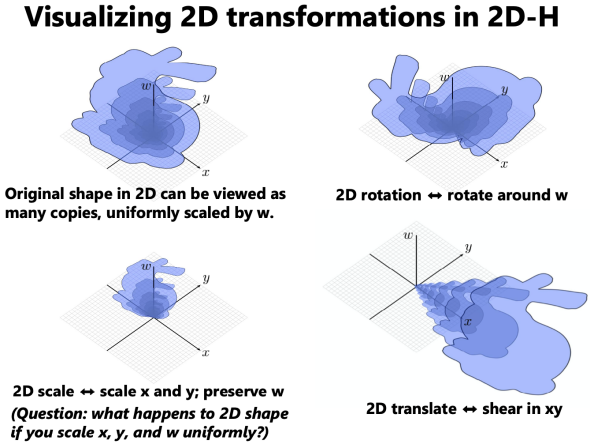
A linear transform can always be regarded as Ax, where A is a matrix and x the vector to be transformed. One example is the change of coordinate systems. To display x in the base $\{\binom{1}{1}, \binom{-2}{1}\}$, we can use the transform $f(x) = \begin{pmatrix} 1 & -2 \\ 1 & 1 \end{pmatrix} x$.

7.4 2D homogeneous coordinated (2D-H)

The key idea is to "lift" 2D points to a 3D space by appending a 1 to the vector. 2D transforms are represented by 3×3 matrices. For example a 2D rotation can be written as:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

Note that in homogeneous coordinates, translation is a linear transformation! This is because translation in 2D-H coordinates is equivalent to shearing along x and y axes, which is a linear operation.
Also note that now we have a difference between point and vector, where the x_3 corresponding entry in the last matrix row is either set to 1 (vector) or 0 (still 2D point).



7.5 Composition of transforms

We can compose linear transforms (and all that we have seen are linear, at least in 2D-H) via matrix multiplication. This scheme works analogously for 3D and

3D-H coordinates.

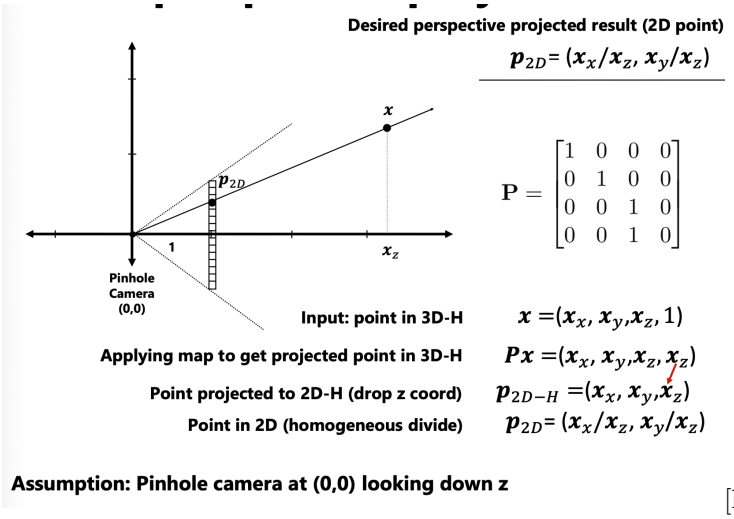
8 Transforms, Geometry and Textures

8.1 Transforms

8.1.1 Simple camera transform

Consider an object at $(10, 2, 0)$. Then for a camera at $(4, 2, 0)$, looking down the x axis we can translate the object's vertex positions by $(-4, 2, 0)$ to get the position relative to the camera. A rotation by $\pi/2$ about the y axis then gives us the position of the object where the camera's view is in direction aligned with the -z axis.

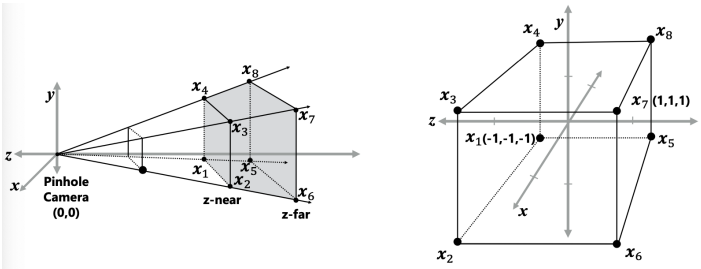
8.1.2 Basic perspective projection



[h]

8.1.3 Changing the shape of the view frustum

The *view frustum* is the region in space that will appear on the screen (it is a truncated pyramid).



Putting it all together, transformation matrix that maps view frustum to unit cube:

$$\mathbf{P} = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zfar+znear}{znear-zfar} & \frac{2\times zfar\times znear}{znear-zfar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

[h]

8.2 Geometry

Geometry is the study of shapes, sizes, patterns and positions. An alternative definition is that geometry is the study of spaces where some quantity (lengths, angles, etc.) can be measured.

8.2.1 Descriptions of geometry

We can describe geometry implicitly, e.g. $x^2 + y^2 = 1$, or also linguistically, e.g. "unit circle", or also explicitly, i.e.$(\cos\theta, \sin\theta)$. There exists no "best" choice for description. Ways for *explicit* encoding are point clouds, polygon meshes, subdivisions. *Implicit* encodings are e.g. level sets, algebraic surfaces, and L-systems. Each choice is best suited to a different task/type of geometry.

Implicit encodings make some tasks easy, e.g. like testing if something is inside/outside. On the other hand, e.g. sampling tasks are pretty hard. Explicit representations give all points in the set directly and therefore make sampling easy, but it's hard to perform the inside/outside test.

8.2.2 Implicit Descriptions

- **Algebraic surfaces**: the surface is the zero set of a polynomial in x,y,z, e.g.

$$x^2 + y^2 + z^2 \text{ for a ball}$$

- **Constructive solid geometry**: since it's very hard to come up for polynomial descriptions, e.g. for a car, we can build up complicated shapes via boolean operations of other shapes.

- **Distance functions**: a distance function gives distance to the closest point on an object.

- **Level set methods**: implicit surfaces have some nice features (e.g. merging, splitting). But it's hard to describe complex shapes in closed form. The alternative is to store a grid of values approximating a function. The surface is found where interpolated values equal zero. This provides much more explicit control over shape. Level sets encode e.g. medical data (CT, MRI).

- **L-systems**: geometry governed by simple rule, the end result has some interesting, useful and/or surprising properties. An L-system is a set of grammar rules, plus a *start symbol* and *semantics*. An example is:

  $$\text{rule 1: A} -> \text{BBC, rule 2: B} -> \text{abA,}$$
  $$\text{then with AAA} -> \text{BBCBBCBBC} -> \text{abACabACabAC} -> ...$$

  We can therefore recursively build up structures like trees. It can be deterministic but also based on probabilistic models.

Pros & Cons of implicit representations

- Pros
  - inside/outside easy to determine
  - distance surface and other queries also easy
  - easy to handle changes in topology (e.g. fluid merging)
- Cons
  - expensive to find all points in the shape (e.g. for drawing)
  - may be difficult to model complex shapes

8.2.3 Explicit Descriptions

- **Point cloud**: simplest representation as a list of all points (x,y,z). Often augmented with *normals*. Easily represent any kind of geometry. Useful for large datasets (>> 1 point/pixel). Difficult to draw in undersampled regions. Hard to do processing / simulation.

- **Polygonal mesh**: store vertices and polygons (most often triangles or quads). Easier to do processing/simulation, adaptive sampling. Perhaps most common representation in graphics. $M = \langle V, E, F \rangle$, where F is the set of faces. Every edge belongs to at least one polygon. The intersection of two polygons in M is either empty, a vertex, or an edge. The *boundary* is the set of all edges that belong to only one polygon. Note the difference between *geometry* (vertex locations) and *topology* (how vertices are connected).

- **Mesh data structures**: allow rendering operations and geometry queries (e.g. what are the vertices of a particular face). They can be stores as *triangle list* (simple, no connectivity, redundancy) or as *indexed face set*, where each triangle only stores an index for a vertex list (no redundancy). Even with indexed face list geometric queries and mesh modifications are costly. This is addressed by more exotic data structures such as half edge data structure.

8.2.4 Sources of geometry

One source is acquiring real-world objects via *3D scanning*. We can also use *digital 3D modeling* to obtain objects.

Subdivision surfaces are an explicit description that smooth out a control curve. For that we insert a new vertex at each edge midpoint. We then update vertex positions according to a fixed rule. For careful choice of the averaging rule, this yields a smooth curve. The procedure is as follows:

1. Start with coarse polygon mesh
2. Subdivide each element
3. Update vertices via local averaging
4. Apply rule, e.g. Catmull-Clark (quads) or Loop (triangles)

Surface normals may be used for shading. *N-dot-L lighting* is the most basic way to shade a surface: take the dot product of unit surface normal (N) and unit direction to light (L), return $\max(0, dot(N, L))$. A common abstraction is infinitely bright light source "at infinity". All light directions (L) are therefore identical.

Sampling color of a point x within a triangle that has its color sources $(1, 0, 0)$ for red, $(0, 1, 0)$ for green, and $(0, 0, 1)$ in its vertices. What is then x's color? We interpolate via *barycentric coordinates*. For that we note that in a triangle with vertices a, b and c, the edges b-a and c-a form a non-orthogonal basis for points in the triangle. Therefore we can write x as

$$x = a + \beta(b - a) + \gamma(c - a) = \alpha a + \beta b + \gamma c, \text{ with } \alpha + \beta + \gamma = 1.$$

There exists also an alternative computation that makes use of the relative size of the subtriangles defined by the edges between the vertices and x.
Due to perspective projection, barycentric interpolation of values in screen XY does not correspond to values that vary linearly on the original triangle. Attribute values must be interpolated linearly in 3D object space. For a perspective-correct interpolation we use the following recipe:

Attribute values (f) vary linearly across triangle in 3D. Due to perspective projection, f/z varies linearly in screen coordinates, and not f directly.

Basic recipe:

- **Compute depth z at each vertex**
- **Evaluate Z:=$1/z$, P:=f/z at each vertex**
- **Interpolate Z and P using standard (2D) barycentric coordinates**
- **At each *fragment*, divide interpolated P by interpolated Z to get f**

Works for any surface attribute f that varies linearly across triangle: e.g., color, depth, texture coordinates

Texture coordinates define a mapping from surface coordinates (points on triangle) to points in texture domain. Note that we may also get *aliasing* due to undersampling textures, because uniformly distributed sample positions in screen space may not be the same positions in texture space and we therefore introduce larger distances.

Filtering textures: in *minification* an area of screen pixel maps to a large region of texture (filtering required/averaging). One texel then corresponds to far less than a pixel on the screen. An example for minification is when a scene object is very far away. The texture map is too large, it contains more details than the screen can display.
Magnification means that an area of screen pixel maps to a tiny region of texture (interpolation required). One texel maps to many screen pixels. An example for magnification is when a camera is very close to a scene object. The texture map is too small.
Mipmap is a technique that prefilters a texture structure in different levels to remove high frequencies. Texels at higher levels store the integral of the texture function over a region of texture space (downsampled images). Texels at higher levels represent low-pass filtered versions of the original texture signal.

9 Graphics Pipeline - The Rasterization Pipeline

9.1 Handling Occlusion and Composition

We already the main parts of the pipeline. We know how to draw a triangle (by sampling), how to represent geometry and transformations, and how perspective projection and texture sampling works. Now we put it all together in an *end-to-end rasterization pipeline.*

9.1.1 Handling occlusion using depth-buffer (Z-buffer)

We want to compute the depth of sampled points on a triangle. For that we interpolate it just like any other attribute that varies linearly over the surface of the triangle. For each coverage sample point, the depth-buffer stores the depth of the closest triangle at this sample point that has been processed by the renderer so far.
The initial state of the depth buffer before rendering any triangles is that all samples store the farthest distance. The greyscale value of each sample point indicates its distance (black = small distance, white = large distance).

To solve the occlusion problem, we process overlapping triangles one-by-one, updating the depth- and color-buffer contents, where a triangle's color is only seen if its the depth-buffer entry of the same index corresponds to the triangle.

Note that the depth-buffer algorithm also handles *interpenetrating surfaces*. The occlusion test is based on the depth of the triangles at a given sample point. The relative depth of triangles may be different at different sample points. The algorithm also works with *super sampling*. The occlusion test is per sample, not per pixel! We even get an anti-aliasing effect if we have e.g. 4 sample points per pixel and can have a mixture of two colors in a pixel.

Summary: occlusion using a depth buffer

- Store one depth value per coverage sample (not per pixel!)
- Constant space per sample; implication: constant space for depth buffer
- Constant time occlusion test per covered sample; read and write of depth buffer if "pass" depth test; just a read if "fail"
- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point
- Range of depth values is limited. That's why the near and far planes are used in defining the view frustum!

9.1.2 Compositing

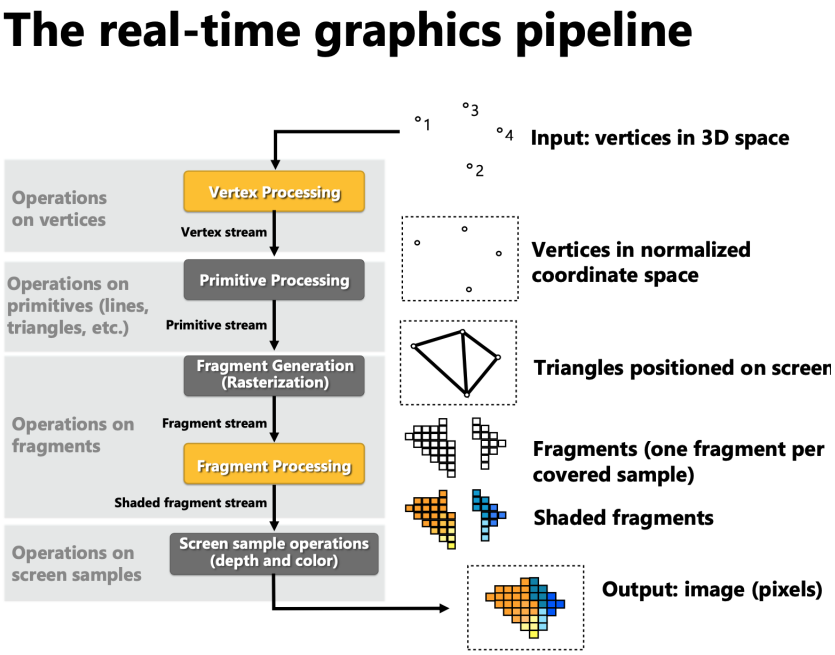
We still have to think about *semi-transparent* objects. For that matter, we use an additional *alpha-channel* of the image (RGBA), which represents the *opacity* (fully opaque $\alpha = 1$, fully transparent $\alpha = 0$).
With this we can define an "over"-operator, i.e. a composition of an image B with opacity α_B over an image A with opacity α_A. Note that the over operator is **not commutative**. Let $A = [A_r A_g A_b]^T, B = [B_r B_g B_b]^T$. Then the composited color $C = \alpha_B B + (1 - \alpha_B)\alpha_A A$, which represents the appearance of semi-transparent B + what B lets through times the appearance of semi-transparent A. The opacity of C is given as $\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$. Note that if

we use a premultiplied α, such that we obtain an A' and B', then we reduce the operations from 2 mult and 1 add to just 1 mult and 1 add: $C' = B' + (1 - \alpha_B)A'$.

Rendering a mixtuer of opaque and transparent triangles

1. Render opaque surfaces using depth-buffered occlusion (if pass depth test, triangle overwrites value in color buffer at sample)
2. Disable depth buffer update, render semi-transparent surfaces in back-to-front order. If depth test passed, triangle is composited OVER contents of color buffer at samples.

9.2 End-to-end rasterization pipeline



How to draw triangles:

1. Transform triangle vertices into camera space.
2. Apply perspective projection transform to transform triangle vertices into normalized coordinate space.
3. Discard triangles that lie complete outside the unit cube (culling). They are off screen, don't bother processing them further. Clip the triangles that extend beyond the unit cube to the cube. Note that clipping may create more triangles.

4. Transform vertex (x, y) positions from normalized coordinated into screen coordinates (based on screen width and height).

5. Triangle preprocessing: compute triangle edge equations and compute triangle attribute equations.

6. Sample coverage, evaluate attributed Z, u, v at all covered samples.

7. Compute triangle color at sample point (color interpolation, sample texture map, or more advanced shading algorithms)

8. Perform depth test (if enabled) and update depth value at covered samples (if necessary).

9. Update color buffer (if depth test passed).

OpenGL/Direct3D are examples of such graphics pipelines, though several stages of the modern OpenGL pipeline are omitted here.

9.3 Shader programs

Shader programs define the behavior of the vertex processing and fragment processing stages. The shader function executes once per fragment and outputs a color of the surface at a sample point that corresponds to the fragment. It also modulates the surface albedo (Rückstrahlvermögen) by incident irradiance (incoming light). For shader programs one uses GPUs that are specialized processors for executing graphics pipeline computations.

9.4 Rasterization Pipeline Summary

- Occlusion is resolved independently at each screen ample using the depth buffer
- Alpha compositing for semi-transparent surfaces
  - Premultiplied alpha form simply repeated composition
  - "Over" compositing operations is not commutative: requires triangles to be processed in back-to-front (or front-to-back) order
- Graphics pipeline:
  - Structures rendering computation as a sequence of operations performed on vertices, primitives (e.g., triangles), fragments, and screen samples
  - Behavior of parts of the pipeline is application-defined using shader programs
  - Pipeline operations implemented by highly optimized parallel processors and fixed-function hardware (GPUs)

9.5 Ray Casting

9.5.1 Basic ray casting algorithm

A sample is a ray in 3D. Coverage is is defined as the ray hitting a triangle (ray-triangle intersection test). Occlusion is done by closest intersection along the ray.

9.5.2 Rasterization vs. Ray Casting

- Rasterization
  - Proceeds in triangle order
  - Most processing is based on 2D primitives (3D geometry projected into screen space)
  - Store depth buffer (random access to regular structure of fixed size)
  - Loop over primitives
- Ray Casting
  - Proceeds in screen sample order. Never have to store depth buffer (just current ray). Natural order for rendering transparent surfaces (process surfaces in the order that they are encountered along the ray: front-to-back or back-to-front).
  - Must store entire scene (random access to irregular structure of variable size: depends on complexity and distribution of scene).
  - Loop over screen pixels

Conceptually, compared to rasterization approach, ray casting is just a reordering of loops + math in 3D. Rasterization and ray casting are two approaches for solving the same problem: determining *visibility*.

9.5.3 Ray Tracing

Ray tracing is a more general mechanism for answering visibility queries. In *recursive ray tracing* one shoots "shadow" rays toward the light source from points where camera rays intersect the scene. When a ray hits a surface, additional rays may be cast because of reflection, refraction, and shadow. If unconcluded, the point is directly lit by the light source.
Reflections can be either *specular, diffuse or glossy*. Refraction is governed by *Snell's law*:

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2},$$

where n_i is the refraction factor of the medium.

10 Light, Color and the Rendering Equation

We need a more precise way to describe and measure the light that arrives at the sensor through the pinhole camera.

10.1 Basics of Light and Color

Light is oscillating electric and magnetic field, where the frequency determines the color of light. The wavelength and frequency are related via the speed of light, i.e. $c = \lambda f$. What we call color is the *visible spectrum* of light.

10.1.1 Additive vs. Subtractive Models of Light

Instead of describing how much of and which wavelength of light is produced (by heat, fusion, etc., which is called *additive model*), we can also describe light with the help of an *absorption spectrum*, i.e. how much light is absorbed (e.g. turned into heat). The emission spectrum is intensity as a function of frequency. The absorption spectrum is the fraction absorbed as a function of frequency.
Many phenomena we care can be described with this. For example, *reflection* is a emission times absorption. A light source that has an emission spectrum $f(\lambda)$ and a surface with reflection spectrum $g(\lambda)$ results in an intensity that is the product of f and g.

10.1.2 Perception of Electromagnetic Radiation

The eyes photoreceptor cells can be distinguished in *rods* and *cones*. Rods are primary photoreceptors for dark viewing conditions (we have approx. 120 million of them), cones (S,M, and L type, 64 % are L cones, 32 % are M cones) are for high-light viewing conditions (approx. 6-7 million). The highest density of cones is in the fovea, i.e. the best color vision is at the center of where the human is looking.

10.1.3 Additive and Subtractive Color models

Just like emission and absorption spectra, we have additive and subtractive color models. Additive models are used for combining colored lights, e.g. RGB. SUbtractive models are used for combining paint colors, e.g. CMYK. Other common color models are HSV (dimensions correspond to natural notions of "characteristics" of color) and Y'CbCr (Y': perceived luminance (intensity), Cb: blue-yellow deviation from gray, Cr: red-cyan deviation from gray).

10.2 Encoding and Quantifying Colors

A common encoding is 8bpc hexadecimal values, e.g. $\#ff6600$ for RGB. We can quantify measurements by *radiometry*, which is a system of units and techniques for measuring electromagnetic radiation. When it comes to measuring, we would like to record some "amount of energy" that arrives. One idea is *radiant energy*,

where we measure the total number of hits that occur anywhere in the scene, over the complete duration of the scene. Another option is *radiant flux*, where we measure hits per second. For illumination phenomena at the level of human perception, it is usually safe to assume that this works well. Another way of measuring is called *Radiance Flux Density* a.k.a. *irradiance*, which counts hits per second and per unit area. To make images, we also need to know where the hits occured. From this point of view, our goal in image generation is to estimate the irradiance at each point of an image (or the total radiant flux per pixel).
Given a sensor with area A, we consider the average flux over the entire sensor area as $\frac{\Psi}{A}$, where Ψ is the flux (energy per unit time in Watts received by the sensor). We can also go the other way and integrate over the flux to obtain the energy $Q = \int_{t_o}^{t_1} \Psi(t)dt$. The irradiance (E) is given by taking the limit of flux as sensor area becomes tiny: $E(p) = \lim_{\Delta \to 0} \frac{\Delta \Psi}{\Delta A} = \frac{d\Psi(p)}{dA}$. To quantify color, we need a *spectral power distribution* that describes the distribution of energy by wavelength.

10.3 Directional Distribution

10.3.1 Angles

In general, we need to think about the directional distribution of irradiance. Remember that in a circle the *angle* is the ratio of a subtended arc length on the circle to the radius, i.e. $\theta = \frac{l}{r}$. A circle has 2π radians.
In a sphere, a *solid angle* is the ratio of an subtended area on the sphere to the radius squared, i.e. $\Omega = \frac{A}{r^2}$. A sphere has 4π steradians. The differential solid angle is given by $d\omega = \frac{dA}{r^2} = \sin(\theta)d\theta d\psi$, i.e. a tiny area on the unit sphere. Integrating the differential solid angle over the unit ball gives us 4π.

10.3.2 Measuring Illumination: Radiance

We will use ω to denote a direction vector of unit length. Radiance is the solid angle density of irradiance. In other words, radiance $L(p, \omega) = \frac{dE_\omega(p)}{d\omega}$ is energy along a ray defined by origin point p and direction ω. Note that we have to break energy down to this granularity, since once we have radiance, we have a complete description of the light in an environment.
We need to distinguish between incident radiance and exitant radiance functions at a point on a surface. In general $L_i(p, \omega) \neq L_o(p, \omega)$.

Radiance is a fundamental quantity that characterizes the distribution of light in an environment. Radiance is the quantity associated with a ray and rendering is all about computing the radiance. Radiance is invariant along a ray in vacuum. A pinhole camera measures the radiance.
We compute the flux per unit area on a surface by integrating over the hemi-

sphere (contributing light arriving from a certain direction):

$$E(p,\omega) = \int_{H^2} L_i(p,\omega)\cos\theta d\omega.$$

10.4 The Rendering equation

The core functionality of a photorealistic renderer is to estimate the radiance at a given point and in a given direction. This is summed up by the rendering equation (Jim Kajiya):

$$L_o(p,\omega) = L_e(p,\omega_0) + \int_{H^2} f_r(p,\omega_i \to \omega_o) L_i(p,\omega_i)\cos\theta_i d\omega_i,$$

where L_o is the outgoing/observed radiance at the point of interest in the direction of interest. L_e is the emitted radiance (e.g. light source), f_r is the scattering function and L_i describes the incoming radiance at an angle between the incoming direction and normal (Lambert's law).
The key challenge is to evaluate the incoming radiance, where we have to compute yet another integral, i.e. the rendering equation is recursive. In practice one uses *Monte-Carlo integration* to solve it approximately.
The *scattering function* describes how reflection of light affects the outgoing radiance. Some basic reflection functions are ideal specular (perfect mirror), ideal diffuse (uniform reflection in all directions), glossy specular (majority of light distributed in reflection direction), and retro-reflective (reflects light back towards source. The choice of the reflection function determines the surfaces appearance.
Many things can happen to a photon at a surface. It can bounce off, transmit through, bounce around inside, get absorbed and re-emitted, etc. The scattering of light can be modeled by the *Bidirectional reflectance distribution function* (BRDF). The BRDF encodes behavior of light that bounces off the surface. Given an incoming direction ω_i, it describes how much light gets scattered in an outgoing direction ω_o. Note that the integral over the scattering function is ≤ 1 (since energy must preserve).
Note that the visual characteristics of many surfaces are caused by light entering at different points than it exits. This violates a fundamental assumption of the BRDF.

11 Raytracing

11.1 Basic Ray casting algorithm

We sample a ray in 3D and test triangle coverage, i.e. if the ray hits the triangle. Occlusion is handled via searching for the closest intersection along the sample ray.

11.2 Rasterization vs. Ray casting

- Rasterization:
  - Proceeds in triangle order
  - Most processing is based on 2D primitives (3d geometry projected into screen space)
  - Store depth buffer (random access to regular structure of fixed size)
  - "Loop over primitives"
- Ray casting
  - Proceeds in screen sample order. We never have to store the depth buffer (just the current ray) and we process the natural order for rendering transparent surfaces (process surfaces in the order they are encountered along the ray: front-to-back or back-to-front)
  - Must store entire scene (random access to irregular structure of variable size: depends on complexity and distribution of scene
  - "Loop over pixels"

11.2.1 Similarities of the approaches

Conceptually, compared to the rasterization approach, ray casting is just a reordering of loops and some math in 3D. Both try to determine *visibility*. Rasterization is more efficient and therefore still preferred for real-time rendering. We can think of rasterization as of an efficient and specialized algorithm for visibility queries, given rays with two specific properties: rays have the same origin, and rays are uniformly distributed over the plane of projection (within specified field of view). These assumptions lead to significant optimization opportunities: we can reduce the triangle-ray intersection to a 2D point-in-polygon test, and the projection to the canonical view volume enables the use of efficient fixed-point math. There exists custom GPU hardware for rasterization.

11.2.2 Shadows

- Rasterization
  - Render scene (depth buffer only) from location of light (pretend there is a camera at light source). Everything "seen" from this point of view is directly lit.
  - Render scene from location of camera. Transform every screen sample to light coordinate frame and perform a depth test (fail = in shadow)
  - Shadows are computed using shadow map (shadow map texture can lead to aliasing)
- Ray tracing

- Recursive ray tracing: shoot "shadow" rays towards light source from points where camera rays intersect scene. If unconcluded, point is directly lit by light source. We therefore check if there is any geometry in the way.
- Correct hard shadows with raytracing. In real life everything is a soft shadow.

11.2.3 Reflections

- Rasterization
  - Environment mapping: place ray origin at location of reflective object, render six views
  - Use camera ray reflected about surface normal to determine which texel in cube map is "hit"
  - Approximates appearance of reflective surface
- Ray Tracing: use recursive ray tracing and get secondary rays. The more rays, the more accurate the final picture. Snell's law describes how rays split on interfaces: $n_1 \sin \delta_1 = n_2 \sin \delta_2$.

11.3 Geometric Queries

11.3.1 Simple Queries

- Closest point to p on 2D line: Use the surface normal, i.e. find t such that $N^T(p + tN) = c$, where $N^T x = c$ describes the line with surface normal. Similar in 3D with plane instead of line.
- Closest point on line segment: Check if point is between endpoints (then closest point to line), otherwise, take closest endpoint of the line segment. Note that we can write the closest point on the line as $a + t(b - a)$. If we solve for t and get $t \in [0, 1]$, then the closest point is inside the segment, i.e. between the endpoints.
- Point-point intersection: just check if the points are equal.
- Point-line intersection: just plug the point into the line equation.
- Line-Line intersection $ax = b, cx = d$ leads to a linear system of equations. Note that for *degenerate* line-line intersection (lines almost parallel), a small change in line normal can lead to a big change in the intersection point, therefore it is unstable and demands special care (e.g. analysis of matrix).

11.3.2 More complex intersections

- Ray-mesh intersection: A ray has a parametric equation, i.e. $r(t) = o+td$, where o is the origin and d its unit direction. To find points where a ray intersects an implicitly defined ($f(x) = 0$) surface, we plug the ray equation $r(t)$ into the x variable of the surface and solve for t (we may get more than one intersection). Finally we plug t back into the ray equation.
- Ray-plane intersection: For ray-plane intersection we run the same procedure: Plug ray into plane equation $N^T x = c$ and solve.
- Ray-triangle intersection: Parameterize triangle given by vertices p_0, p_1, p_2 using barycentric coordinates $f(u, v) = (1-u-v)p_o+up_1+vp_2$. Then plug the parametric ray equation directly into equation for points on triangle, i.e. $f(u, v) = r(t)$, and solve for u, v and t.

11.4 Optimizations for Ray Intersection

We may use the already seen idea of using *bounding boxes* and can check in $O(1)$ complexity if the ray misses a box and therefore all primitives in the scene. To further optimize we may use a *bounding volume hierary (BVH)*, and order it as a tree. That means, larger boxes contain smaller ones where interior nodes in the tree represent a subset of primitives in the scene and leaves contain a list of primitives. Note that the sets can still be overlapping in space. There are $2^N - 2$ ways to partition N primitives into two groups, therefore we have many possible BVHs. A "good" partition partitions into child nodes with equal number of primitives and minimizes the overlap between children (and also avoid empty space). Partitions can be implemented by choosing an axis, then choose a split plane on that axis and partition the primitives by the side of splitting plane their centroid lies. Troublesome cases are where all primitives have the same centroid (and therefore end up in same partition) or where all primitives have the same bbox (ray often ends up visiting both partitions).
An alternative to primitive partitioning is *space partitioning* (grid, K-D tree) that partitions space into disjoint regions (primitives may be contained in multiple regions of space).
Another partition method is the *uniform grid*. We partition the space into equal sized volumes called *voxels* and each grid cell contains primitives that overlap that voxel. This can be very efficient, since we only have to consider intersection with primitives in voxels that the ray intersects. Note that we have a trade-off in grid resolution (too few cells degenerates to brute-force approach, too many incur significant cost traversing along the ray with empty voxels). A good heuristic is to choose the number of voxels proportional to the total number of primitives.
K-D trees recursively partition space via axis-aligned planes. Interior nodes correspond to spatial splits and node traversal can proceed in front-to-back order (unlike BVH, can terminare search after first hit is found). If objects overlap multiple nodes of the tree, then we require the intersection point to be within

current leaf node (primitives may be intersected multiple times by the same ray).

11.5 Summary of Acclerating Geometric Queries

- Primitive vs. spatial partioning:
  - Primitive partitioning: partition sets of objects. Bounded number of BVH nodes, simpler to update if primitives in scene change position
  - Spatial partitioning: partition space, traverse space in order (first intersection is closest intersection), may intersect primitive multiple times
- Adaptive structures (BVH, K-D tree)
  - More costly to construct (must be able to amortize construction over many geometric queries)
  - Better intersection performance under non-uniform distribution of primitives
- Non-adaptive accleration structures (uniform grids)
  - Simple and cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed

12 Introduction to Computer Animation

So far we have increased the complexity of our models (transformations, geometry, materials, lighting), but we have not yet seen how to describe motion. *Animation* is creating the illusion of motion, e.g. by rapidly displaying a sequence of static images that minimally differ from each other.

12.1 Twelve Principles of Animation

1. Squash and Stretch: Defining the rigidity and mass of an object by distorting its shape during an action.
2. Timing: Spacing actions to define the weight and size of objects, and the personality of characters.
3. Anticipation: The preparation for an action.
4. Staging: Presenting an idea so that it is unmistakably clear.
5. Follow Through and Overlapping Action: The termination of an action and establishing its relationship to the next action.

6. Straight Ahead Action and Pose-To-Pose Action: The two contrasting approaches to the creation of movements.
7. Slow In and Out: The spacing of in-between frames to achieve subtlety of timing and movements.
8. Arcs: The visual path of action for natural movement.
9. Exaggeration: Accentuating the essence of an idea via the design and the action.
10. Secondary Action: The action of an object resulting from another action.
11. Appeal: Creating a design or an action that the audience enjoys watching.
12. Solid Drawing: Knowing them can dramatically improve one's ability to create good, strong poses and compose them with well crafted environments.

12.2 Generating Motion

There exist different basic techniques in computer animation to generate motion. Possible options are artist-directed (e.g. keyframing), data-driven (e.g. motion capture) or procedural (e.g. simulation) generation.

12.2.1 Keyframing

The basic idea is that we specify important events only and the computer fills the rest via interpolation/approximation. The interpolation can be done e.g. via piecewise linear interpolation (simple, but rather rough motion) or piecewise polynomials, e.g. in form of splines, where we get better continuity than in linear interpolation.

12.2.2 Splines

A *spline* is any piecewise plynomial function that interpolates data points $(t_i, f_i), i = 0, ..., n$. Interpolates means that the function exactly passes through those values. The only other condition is that the function is a polynomial when restricted to any interval between knots, i.e. for $t_i \leq t \leq t_{i+1}, f(t) = \sum_{j=1}^{d} c_i t^j =: p_i(t)$, where d is the degree. Commonly used splines for interpolation are cubic (d = 3). Piecewise cubics give the exact solution to the elastic spline interpolation problem under assumption of small displacements. More precisely, among all curves interpolating a set of data points, cubic splines minimize the norm of the second derivative. *Runge's phenomenon* describes the attempt to use higher-degree polynomials to obtain higher-order continuity, but this can lead to strong oscillation and worse approximation.

Cubic polynomials have four *degrees of freedom (DOF)* namely the four coefficients (a, b, c, d) describing $p(t) = at^3 + bt^2 + ct + d$. We only need two DOF

to specify the endpoints. To obtain unique solutions, we also match the derivatives at the endpoints. This gives us a linear system of four equations. Only having the tangents agree gives us C^1 *continuity*, if we also agree the curvature (second derivative) then we obtain C^2 *continuity*.
We now list some properties of a good spline:

- Interpolation: spline passes exactly through data points.
- Continuity: at least twice differentiable.
- Locality: moving one control point doesn't affect whole curve.

Natural splines fulfil the first two points, but they lack in locality, since the coefficients depend on a global linear system. Note that natural splines have $4n$ DOFs, and $2n + (n-1) + (n-1) = 4n - 2$ equations.

Hermite Splines are splines where each cubic piece is specified by endpoints and tangents. This is different to natural splines since there the tangents didn't have to match any prescribed value. In Hermite splines they are given, we therefore have $2n + 2n = 4n$ conditions. Hermite splines fulfil interpolation and locality, but they lack in C^2 continuity.

Catmull-Rom Splines are a specialization of Hermite splines, determined by values alone. We do not specify the values of the tangents explicitly, but use the difference of neighbors to define the tangents at interval borders, i.e. the tangent is $u_i := \frac{f_{i+1} - f_{i-1}}{t_{i+1} - t_{i-1}}$. They have the same properties as any other Hermite spline and are commonly used to interpolate motion in computer animation. Catmull-Rom is usually a good starting point.

B-Splines are a kind of spline that satisfy continuity and locality, but sacrifice interpolation. A B-spline is recursively defined as

$$B_{i,1}(t) := \begin{cases} 1, & \text{if } t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

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

Summary of Splines

	INTERPOLATION	CONTINUITY	LOCALITY
natural	YES	YES	NO
Hermite	YES	NO	YES
B-Splines	NO	YES	YES

Example for interpolation:
Animate position and view direction. Each path is a function $f(t) = (x(t), y(t), z(t))$ and each component (x, y, z) is a spline.

13 Rigging, Forward Kinematics and Inverse Kinematics

13.1 Rigging and Deformers

13.1.1 Rigging

Animation Rigs are user-defined mappings between a small number of parameters and the deformations of a high-resolution mesh. We only define some points (like bones) and how the mesh gets deformed. Animations are time trajectories specified for rig parameters.
The simplest type of rig are *blend shapes* (such as blender or other software use). We input a set of meshes M_i with vertices x_i^j with blending weights $\alpha = (\alpha_1, ..., \alpha_n)$ and output a blended mesh $M = \sum_i \alpha_i M_i, $, i.e. $x^j = \sum_i \alpha_i x_i^j$ obtained through interpolation. The keyframes in form of splines specify blending weights over time. The blend shape *sculpting* is usually a manual process.
Note that blend shapes are not ideal if we linearly interpolate between weird positions or rotate, such that we change only a pose.

13.1.2 Cage Based Deformers

Cage based deformers are an alternative to blend shapes. In this model we only have a coarse mesh (cage) and first deform the coarse mesh, and then reconstruct the high-resolution geometry. Working with cages can be very restrictive though, since it is hard to have direct control of small regions, e.g. teeth of a figure. Very often the shape of an object implies a skeleton, which gives us a well-defined structure. In this case the key idea is to just animate the skeleton (much less than DOFs) and have the mesh "follow" automatically. We then have to simulate muscles, fatty tissues, skin, interactions with the environment, etc.,

which can be very demanding computationally. Hence we need faster solutions for interactive applications.

13.2 Forward Kinematics

Forward kinematics refers to the use of the kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters. The *kinematic skeleton* is defined as follows: *Joints* are local coordinate frames. Bones are vectors between consecutive pairs of joints. Each non-root bone is define in a frame of a unique parent. Changes to the parent frame affect all descendent bones. Both skeleton and skin are designed in a rest (dress/bind) pose. We then perform *affine transformations* on the hierarchy. Assume $n+1$ joints, $0, 1, ..., n$ where 0 is the root. Each joint corresponds to a frame. $p(j)$ is the parent of joint j (root parent is defined as -1). Then the frame joint j expressed w.r.t. frame of $p(j)$ is:

$$_{p(j)}R_j = \begin{pmatrix} Rot(j) & t(j) \\ 0 & 1 \end{pmatrix},$$

where $t(j)$ typically comes from the bind pose and *Rot(j)* comes from animation. The transformation from frame j to the world is:

$$_wR_j = T(0)Rot(0)...T(p(j))Rot(p(j))T(j)Rot(j).$$

Each joint rotation $Rot(j)$ has up to 3 DOFs. Translation and scale is also possible, but less common. $_w\bar{R}_j$ is defined in the same way as $_wR_j$, but in rest pose (e.g. default bind pose).

13.3 Skinning

After computing the configuration via forward kinematics, we have to reconstruct the surface mesh. A simple first attempt is called *rigid skinning*, where we assign each vertex to its closest bone and compute the world coordinates according to the bone's transformation. This is fine if the vertices are close to the bones (was used in older video games).
A better attempt with a slight increase in complexity is *Linear Blend Skinning*. We assign each vertex to multiple bones and compute the world combination as *convex combination*. The weights count for the influence of each bone on the vertex. This leads to smoother deformations of the skin. Formally:

$$v = \sum_j \alpha_j \, {}_wR_j \, {}_{\bar{w}}\bar{R}_j{}^{-1} v',$$

where v are the skinned vertex coordinates, α the skinning weights, the bar transformation the vertex coordinates in coordinate frame of bone j, and v' the vertex coordinates in rest pose.
Linear blend skinning may suffer from artifacts called *candy-wrapper effects*, that can be fixed by using auxiliary joints/bones, employing better interpolation schemes for transformations, or using pose-space deformers (PSD).

13.4 Inverse Kinematics (IK)

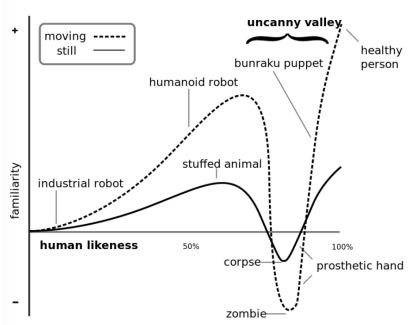
IK is the reverse process to FK that computes the joint parameters that achieve a specified position of the end-effector. The basic idea of *optimization based IK* is that we write down the distance between the final point and the "target" and set up the following objective:

$$f_0(\theta) = \frac{1}{2}(p(\theta) - \tilde{p})^T(p(\theta) - \tilde{p}).$$

We then compute the gradient w.r.t. the angles and go "downhill" from there. We could also constraint the joint angles by introducing a limit on them.

13.4.1 Pitfalls of hand-crafted animation: The Uncanny Valley

In aesthetics, the uncanny valley is a hypothesized relationship between the degree of an object's resemblance to a human being and the emotional response to such an object. The concept suggests that humanoid objects which imperfectly resemble actual human beings provoke uncanny or strangely familiar feelings of eeriness and revulsion in observers.



13.4.2 Full-Body Motion Capture

Motion capture provides sparse signals (e.g. marker trajectories) from which fully body motion needs to be reconstructed. Often this is posed as an optimization problem, e.g. via inverse kinematics.

14 Introduction to Physically-based Animation and ODEs

14.1 Kinemtics, Dynamics and the Animation Equation

Kinematics is the branch of mechanics concerned with the motion of objects without reference to the forces which cause the motion. On the other hand, *dynamics* is the branch of mechanics concerned with the motion of bodies under the action of forces. Remember that rasterization and path tracing gave

approximate solutions to the rendering equation. We now study the *animation equation*, which covers a large spectrum of physical materials and phenomena (solids, fluids, elasticity, plasticity, magnetism, etc.) and leverages tools from computational physics (e.g. numerical integration).

14.2 Equation of Motion

The *Equation of Motion (EoM)* is $F = ma$, where F is the force, m the mass and a the accleration. Every system has a *configuration* $q(t)$ and also a *velocity* $\dot{q} := \frac{d}{dt}q$. It also has some kind of *mass* M and there are probably some *forces* F. We may introduce some constraints like $g(q, \dot{q}, t) = 0$. With this in mind, we can write Newton's 2nd law as $\ddot{q} = F/m$. *Accleration* is the 2nd time derivative of configuration and ultimately we want to solve for the configuration q.

Often we describe systems with many moving pieces. We collect them all into a single vector of *generalized coordinates* $q = (x_0, ..., x_n)$, where we can think of q as a single point moving along a trajectory in $\mathbb{R}^n$. This way of thinking naturally maps to the way we actually store equations on a computer: all variables are often "stacked" into a big vector and handed to a solver. The velocity of q is just the time derivative of the generalized coordinates, i.e. $\dot{q} = (\dot{x}_0, ...\dot{x}_n)$. This also holds for generalized masses and forces in a similar way.

14.3 Ordinary Differential Equations

Many dynamical system can be described via an ordinary differential equation (ODE, involves derivatives w.r.t. only one variable):

$$\frac{d}{dt}q = f(q, \dot{q}, t),$$

where f is the velocity function and $\frac{d}{dt}q$ the change in configuration over time. Note than we can write a second order ODE as two first order ODEs, by introducing a "dummy" variable for the first order derivative, e.g.

$$\ddot{q} = F/m, \text{ for which we get two first order ODEs: } \dot{q} = v, \dot{v} = F/m.$$

Note that we still need an *initial value* to fix the solution.

A not so easy example is the *n-Body Problem*, e.g. planets gravitation. As soon as $n \geq 3$, no closed form exists and we get chaotic solution.

Particle Systems model complex phenomena as large collection of particles. Due to the simplicity of the equation for each particle we can be easily scale to many particles. Many particles may require fast hierarchical data structures (kd-tree, BVH, etc.). Examples for particle systems are flocking (birds), crowd simulation or granular materials.

14.4 Solving ODEs: Numerical Integration

We want to obtain a function $q(t)$, given $q(0)$ and $\dot{q}(q, \dot{q}, t)$. The key idea is that we replace the time-continuous function $q(t)$ with samples q_t (also easy to get using a truncated taylor expansion). We go from $\frac{d}{dt}q(t) = f(q(t))$ and want to obtain

$$\frac{q_{k+1} - q_k}{\tau} = f(q),$$

where q_{k+1} is the new configuration for which we want to solve for. The question is how to evaluate the velocity function f at the point q.

14.4.1 Forward Euler

The simplest (Runge-Kutta) scheme for that is *forward euler*, where we evaluate the velocity at the current configuration. The new configuration can then be written explicity in terms of known data, i.e.

$$q_{k+1} = q_k + \tau f(q_k).$$

This intuitive method just walks a tiny bit in the direction of the velocity. Unfortunately, it is **not very stable**. In practice we need very small timesteps ("stiff" ode).

14.4.2 Backward Euler

In the *backward euler* method we evaluate the velocity at the next configuration. The new configuration is then implicitly defined and we must solve for it:

$$q_{k+1} = q_k + \tau f(q_{k+1}).$$

This is much harder to solve, since in general f can be very nonlinear. The benefit is that backward euler is **unconditionally stable** for linear ODEs. Note that empirically backward euler exhibits *numerical damping* (the decrease in amplitude of the numerical solution is purely an artifact of the numerical method).

14.4.3 Symplectic Euler

Symplectic Euler is the alternative, where we update the velocity using the current configuration, and update the configuration using the new velocity. This resembles forward euler under velocity, and backward euler under configuration. The method is easy to implement and energy is conserved almost exactly, and therefore often used in practice.

15 Partial Differential Equations

Remember that an ODE implicitly describes a function in terms of its time derivatives. PDEs also include spatial derivatives. An example is the heat equation (temperature of a particle on a wire): $\frac{dT}{dt} = \alpha \frac{\delta^2 T}{\delta x^2}$.

15.1 Definition and Anatomy of a PDE

We want to solve for a function of time and space, i.e. $u(t, x)$. The function is given implicity in terms of derivatives, e.g. $\ddot{u}$ or $\frac{d}{dt^3}u$.
There are two types of PDEs. In *nonlinear* PDEs, such as *Burgers' equation* ($\dot{u} + uu' = au''$), different derivatives (space and time) are combined. In *linear* PDEs, such as the diffusion equation ($\dot{u} = au''$), there is only one type of differential in each subterm.
The *order* of the PDE describes how manz derivatives in space and time we have, e.g. Burgers' equation is of order 2, since we have a second order derivative in space. Higher order PDEs are harder to solve, especially if they are nonlinear as well.

15.2 Model Equations

The fundamental behavior of many important PDEs is well-captured by three model linear equations. We first introduce the *Laplace operator*, before we go over to the model equations.
The *Nabla operator* is defined as $\nabla = (\frac{\delta}{\delta x_1}, ..., \frac{\delta}{\delta x_d})^t$, i.e. the gradient of a function. The Laplace operator is defined as $\Delta = \nabla \cdot \nabla = \sum_{k=1}^{d} \frac{\delta^2}{\delta x_k^2}$, i.e. the multiplication of the operator is element-wise. The semantics of the Laplace operator is the divergence of the gradient. Alternative interpretations are the trace of the hessian, the sum of second derivatives or the "average" curvature.

15.2.1 Laplace Equation

The first model equation we take a look at is the (elliptic) *Laplace equation*:

$$\Delta u = 0,$$

which searches for the smoothest function interpolating the given boundary data. It is quite easy to solve numerically. Conceptually each value is the average of its "neighbors", which makes elliptic PDEs are very robust to errors. If we discretize the Laplacian operator (either as grid or triangle mesh), at the solution of $\Delta u = 0$, each value is the average of its neighboring values. For a grid with center a and neighbors b, c, d and e, we have $a = 1/4(b + c + ? = e)$, where we get to choose d. This is the data we want to interpolate. There are two basic boundary conditions: *Dirichlet* (boundary data always set to fixed values), and *Neumann* (specify derivative (difference) across boundary). One may also used mixed conditions.

15.2.2 Heat Equation

The second model equation is the (parabolic) *heat equation*

$$\dot{u} = \Delta u,$$

which describes how an initial distribution of heat spreads out over time. It is not as easy so solve. After a long time, the solution will be the same as from the Laplace equation. Examples are damping or viscosity in physical systems. When solving the heat equation, e.g. for forward Euler we get $u^{k+1} = u^k + \Delta u^k$, where we can use e.g. the grid discretization of the Laplacian.

15.2.3 Wave Equation

The third model equation is the (hyperparabolic) *wave equation*

$$\ddot{u} = \Delta u,$$

which describes how the wavefront evolves over time, e.g. if we throw a rock into a pond. It is an advanced topic to solve. It does not have a steady state solution and errors made at the beginning will persist for a long time. Solving the wave equation is not much different to solving the heat equation. We first convert it to two first order (in time) equations, i.e. $\dot{u} = v, \dot{v} = \Delta u$ and then evaluate the spatial derivative, i.e. $\frac{u^{k+1} - 2u^k + u^{k-1}}{\tau^2} = \Delta u^k$. WEe then integrate forward in time using, for example symplectic Euler. Note that this is only one way to solve this PDE, there are many more.

In general, nonlinear, hyperbolic and high-order equations are really hard to solve.

15.3 Solving PDEs

Like ODEs, many interesting PDEs are difficult or impossible to solve analytically. Instead we must use numerical integration. The basic strategy is as follows:

1. Pick a spatial discretization
2. Pick a time discretization (forward/backward Euler)
3. Run a time-stepping algorithm (as with ODEs), which ultimately generates the animation

There are two basic ways to discretize space. The *Lagrangian method* is particle based, i.e. tracks moving particles and reads what they are measuring. The *Eulerian method* is grid-based, i.e. records the temperature at fixed locations in space. Lets compare the two methods and its tradeoffs:

- Lagrangian
  - \+ Conceptually easy (like a polygon soup)
  - \+ Resolution/domain is not limited by a grid
  - − Good particle distribution can be tough
  - − Finding neighbors can be expensive

- Eulerian
  - + Fast and regular computation
  - + Easy to represent, e.g. smooth surfaces
  - − Simulation "trapped" in grid
  - − Grid causes "numerical diffusion" (blur)
  - − Need to understand PDEs

In practice, there is no reason to not mix the both methods. Make sure to pick the right tool for the right job.