Visual Computing

by dcamenisch

1 Introduction

This document is a summary of the 2022 edition of the lecture *Visual Computing* at ETH Zurich. I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. If you spot any mistakes or find other improvements, feel free to open a pull request at `https://github.com/dcamenisch/ethz-summaries`. This work is published as CC BY-NC-SA.

Computer Vison

2 The Digital Image

An image is simply a continuous function over 2 or 3 variables (XY-coordinates and possibly time). Usually we use brightness as the value of the function, but other physical values can also be used. For a computer this is just a collection of numbers, but instead of continuous values we have discrete. Note that in real life images are never completely random and almost always contain some structure. It is important to know that **pixels are not little squares**, they are point measurements.

When taking a picture with a digital camera, we can encounter various problems, e.g.:

- Transmission Interference
- Compression Artefacts
- Spilling
- Sensor Noise

2.1 Sampling

When taking an image, we are sampling such a continuous. When trying to reconstruct the original function, we can encounter undersampling, i.e. when we loose information due to a too low amount of sampling points.
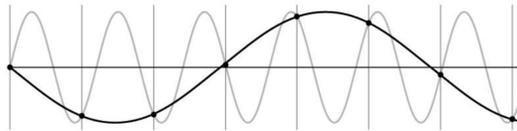

Due to undersampling, the result can not be distinguished from a lower or a higher frequency wave. Signal disguised as other frequencies is also called **aliasing**.

Nyquist-Shannon Sampling Theorem
For sine waves we have to sample at half the wave length. This corresponds to double the frequency, we also call this the **Nyquist Frequency**.

2.2 Quantization

Another problem we have to deal with is **quantization**, since the real valued function will get digital (integer) values, it is lossy. Compared to sampling which lets us reconstruct the original function. Simple quantization uses equally spaced levels with k intervals.
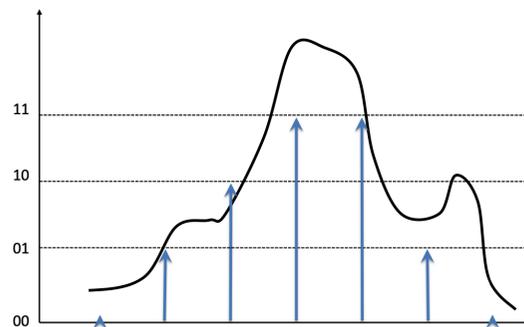
2.3 Image Properties

Image resolution is divided into two parts:

- Geometric Resolution: How many pixels per area
- Radiometric Resolution: How many bits per pixel

2.4 Noise

When taking pictures we can almost always encounter some noise. A common way to model this is additive gaussian noise:

$$I(x, y) = f(x, y) + c, \qquad c \sim \mathcal{N}(0, \sigma^2)$$

The signal to noise ratio (SNR) is an index of image quality:

$$\text{SNR} = \frac{F}{\sigma}, \qquad F = \frac{1}{XY} \sum_{x=1}^{X} \sum_{y=1}^{Y} f(x, y)$$

The usefulness for this metric can vary drastically depending on the type of image (dark images will have a higher SNR compared to bright images). Therefore we introduce peak SNR:

$$\text{PSNR} = \frac{F_{\max}}{\sigma}$$

3 Segmentation

Image segmentation is often viewed as the ultimate classification problem, once solved, computer vision is solved. A complete segmentation of an image is a finite set of disjunct regions $R_1, ..., R_n$, such that $I = \bigcup R_i$.

3.1 Thresholding

Thresholding is a simple segmentation process, that produces a binary image by labelling each pixel in or out of the region of interest. We do this by comparison of the grey level with a threshold value T. Another, better approach can be chromakeying. Hereby we measure the distance from a defined color g:

$$I_\alpha = |I - g| > T$$

One limit of thresholding is that it does not consider image context.

3.2 Segmentation Performance

If we want to choose the best performing segmentation algorithm or determine a good value for T, we need a performance metric. To use automatic analysis, one needs to know the true classification of each test, for this the test images have to be segmented by hand.

One performance metric is the ROC curve. This curve characterizes the error trade-off in binary classification tasks, by plotting the true positive fraction against the false positive fraction. We often choose the operating point on the ROC curve, by assigning cost and values to each outcome:
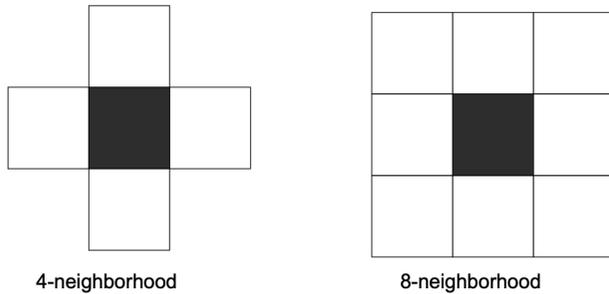
- V_{TN} - value of true negative
- V_{TP} - value of true positive
- C_{FN} - cost of false negative
- C_{FP} - cost of false positive

We then choose the point on the ROC curve with the gradient:

$$\beta = \frac{N}{P} \cdot \frac{V_{\text{TN}} + C_{\text{FP}}}{V_{\text{TP}} + C_{\text{FN}}}$$

3.3 Pixel Connectivity

We try to define which pixels are neighbors.

4-neighborhood   8-neighborhood

A 4 (or 8) connected path between p_1, p_n is a set of pixels such that every p_i is a 4 (or 8) neighbor of p_{i+1}. Now we can defined a region as 4 (or 8) connected if it contains a 4 (or 8) connected path between any two of its pixels.

With this we can introduce **region growing**. We start from a seed point or region and add neighboring pixels that satisfy the criteria defining a region until we can include no more pixels. There are different approaches to selecting the seed and we could also use multiple seeds. For the inclusion criteria we could choose thresholding or a distribution model.

Another criteria is a snake (active contour). While each point along the contour moves away from the seed, it always has to have some smoothness constraints (minimizing energy function).
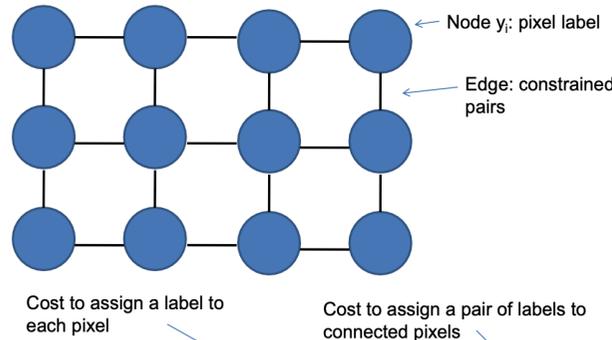
3.4 Distance Measures

Plain background substraction $I_\alpha = |I - I_{bg}| > T$, where I_{bg} is the background image, we get this by fitting a Gaussian (Mixture) model per pixel. Even better would be:

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

Where Σ is the background pixel appearance covariance matrix.

3.5 Markov Random Fields

We can add spatial relations with Markov Random Fields (2D Markov Chains).

← Node y_i: pixel label

← Edge: constrained pairs

Cost to assign a label to each pixel   Cost to assign a pair of labels to connected pixels

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

Using a graph cut algorithm we can determine the optimal segementation. We can further optimize this by using iterated graph cut and k-means for learning the colour distribution (GMM) of the image.
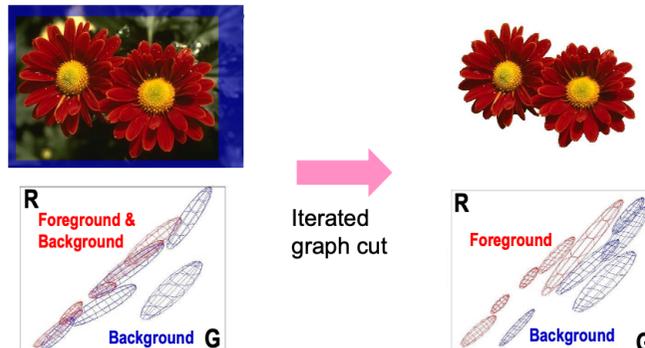
4 Convolution and Filtering

Convolution and filtering are some of the most basic operations of image processing.

4.1 Filtering

Image filtering is the process of modifying pixels in an image base on some function of a local neighborhood of the pixel.

#### 4.1.1 Linear Shift-Invariant Filtering

Linear shift-invariant filtering means using linear combinations of neighbors and doing the same for each pixel (shift-invariant). These filters are often used for low-level image processing, smoothing / noise reduction, sharpening and feature detection. Linear operations can be written as:

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

Here I is the input image, I' the output image, K is the kernel and N is the neighborhood. Operations are shift-invariant if K does not depend on (x,y).
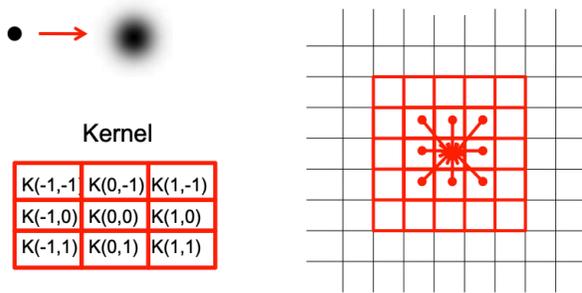
4.2 Correlation

Correlation, e.g. template matching:
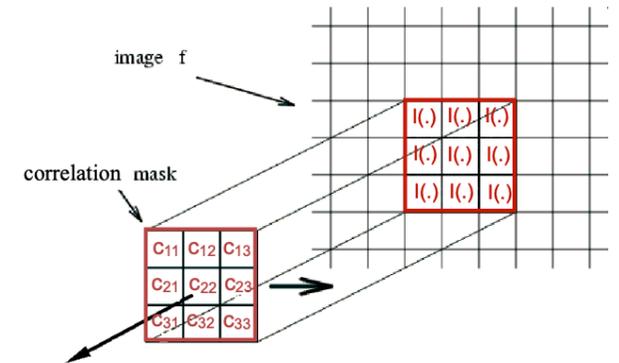
$$I' = K \circ I, \quad I'(x,y) = \sum_{(i,j) \in N(x,y)} K(i,j) I(x+i, y+j)$$

Correlation takes an input image and a weight mask, then each pixel gets "replaced" by the weighted sum of its neighborhood. This can be described as taking multiple input location and writing one output location.

4.3 Convolution

Convolution, e.g. point spread function:

Kernel

K(-1,-1)	K(0,-1)	K(1,-1)
K(-1,0)	K(0,0)	K(1,0)
K(-1,1)	K(0,1)	K(1,1)

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

This is similar to the correlation, but **the kernel is reversed**.

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

If $K(i,j) = K(-i,-j)$ we call the operations dual.

Convolution can be described as taking one input location and writing multiple output location, the opposite of the correlation. By default we use convolution for filtering. An example for a kernel would be:

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This kernel is used for sharpening by accentuating differences with the local average. Another example would be:

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

This kernel looks for differences in the horizontal direction, this corresponds to finding vertical edges.

4.3.1 What about the Edges?

If we apply our filters to images, we need to **deal with the edges separately**. This is due to our window falling off the edge of the image. There are different techniques to deal with this problem:

- Extend the image with black border
- Wrap the kernel around the edges
- Copy out the edge
- Mirror the image at the edge
- Vary filter near the edge

4.3.2 Separable Kernels

A kernel is separable, if it can be written as $K(m,n) = f(m)g(n)$. This means that the kernel can be separated into a function for the first coordinate and another for the second coordinate. If this is the case we can apply the separated functions individually to the image.

A filter is separable, if the kernel matrix has rank 1.

4.3.3 Gaussian Kernel

The idea of the **Gaussian Kernel** is to weight the contributions of neighboring pixels by nearness.
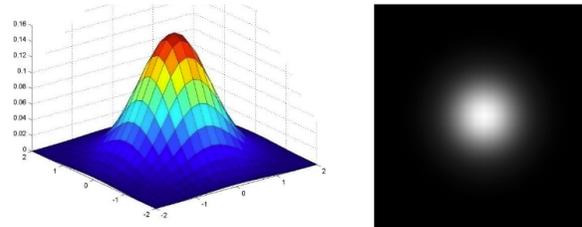
$$G_\sigma = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

We can use the Gaussian Kernel for image smoothing, the best part being that the kernel is separable. The actual amount of smoothing depends on σ and the window size.

If we repeatedly apply the Gaussian filter, we produce the scale space of an image.

4.3.4 High-Pass Filters

High-pass filters are used to detect areas of the image where a lot is happening (high frequencies). Examples for these are the Laplacian operator K or the high-pass filter K':

$$K = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad K' = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

High-pass filters can be used to perform image sharpening $I' = I + \alpha|K * I|$.

4.3.5 Integral Image

The integral image or summed area table is defined as:

$$\int_0^x \int_0^y I(x,y)dydx$$

It is the sum of all pixels in a rectangle spanned from the origin to a point (x,y). The integral image can be used to speed up calculating the convolution with a constant rectangle.

We can get a high-pass filter from a *normalized* low-pass filter by calculating $I - L = H$.

5 Image Features

Image features are about detecting the location of patterns in images, e.g. edge detection or facial landmarks.

5.1 Template Matching

Given an template t, e.g. template describing an eye, we want to locate an area, in an image s, that best fits this template. Alternatively we could also look for areas that match this template by a certain threshold.

To search for the best match, we try to minimize the mean squared error. This is the same as maximizing the area correlation:

$$r(p,q) = \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} s(x,y) \cdot t(x-p,y-q) = s(p,q) * t(-p,-q)$$

5.2 Edge Detection

For edge detection we need to calculate the gradient magnitude M and the gradient orientation α:

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

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

We have previously seen kernels that can detect horizontal edges. To expand on this, we differentiate the following kernels:

$$\text{Prewitt} \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad \text{Sobel} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

We can also transpose these kernels to detect horizontal edges. From the resulting images (horizontal / vertical edges) we take the log sum squared and use different thresholds to achieve the final result.

5.2.1 Laplacian Operator

The idea behind the Laplacian operator is to detect discontinuities in the second derivative. This corresponds to detecting zero-crossings. The operator is isotropic (rotationally invariant) and can be implemented with one of the following kernels:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This operator is very sensitive to fine details and noise, therefore we might need to blur the image first. Additionally it will respond equally to weak and strong edges, so we want to suppress edges with low gradient magnitude. Blurring and applying the Laplacian operator can be combined into a convolution with Laplacian of Gaussian (LoG). Combining LoG with gradient based threshold delivers the best result.


5.2.2 Canny Edge Detector

The Canny edge detector works by first smoothing the image with a Gaussian filter. Then we compute the gradient magnitude (Sobel, Prewitt, ...) and the angle of the gradient. After this we want to apply non-maxima suppression to the gradient magnitude image. Combining this with double thresholding, to detect strong and weak edge pixels, and rejecting weak edge pixels not connected with strong edge pixels, results in the Canny edge detector.
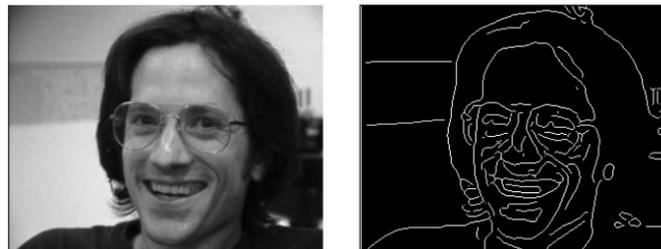

5.3 Hough Transform

Hough transform can be used to find higher order entities in an image, e.g. lines or circles. The Hough transform is a generalized template matching technique. Considering detection of straight lines (y = mx + c), for each edge pixel there are infinitely many possible lines. We plot these possible lines in the 2D space formed by its parameters, we end up with a single line.
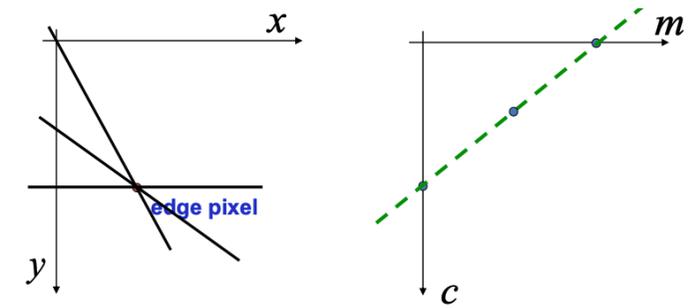

If we do this for multiple edge points and subdivide the parameter space into discrete bins, we can find the bin with the most possible lines. This gives us the detected line.
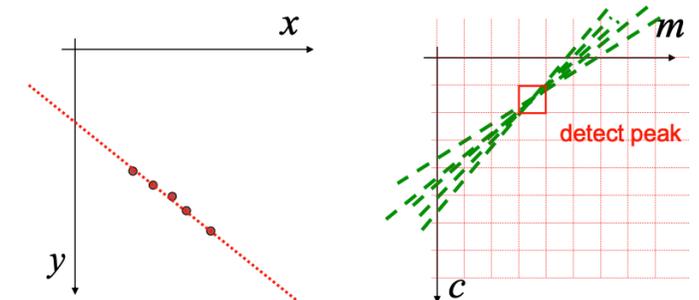
There is a problem with this approach, the parameter space is infinite. To avoid this problem we choose an alternative parametrization, in this case we represent a line as an angle and the distance from the origin. Now the representations in parameter space are not lines but sine waves.

$$p = x \cos\theta + y \sin\theta$$

Again we find the maxima to find our lines.
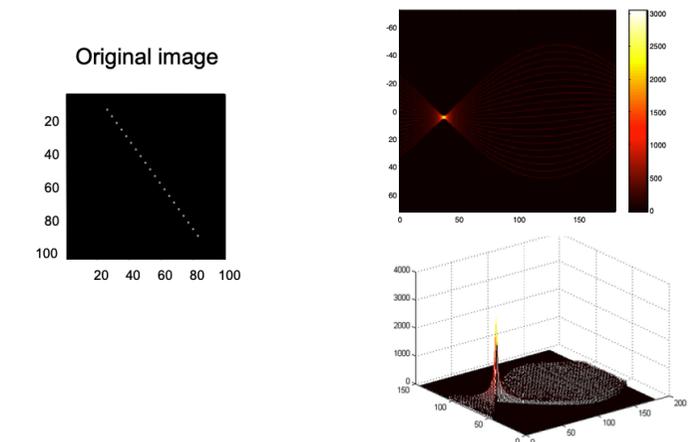
To find multiple lines we do non-maxima suppression and keep every strong peek. To expand this concept to circle detection we simply change the parameter space.

One downside is that two line segments that lie on the same line but are some difference apart cannot be distinguished but appear as a single line.

5.4 Keypoint Detection

We might want to only find corners and not edges. We want this corner localization to be accurate, invariant and robust. We define the following:

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

$$S(\Delta x, \Delta y) = (\Delta x, \Delta y) \ \mathbf{M} \ (\Delta x, \Delta y)^\top$$

Hereby f_x is the horizontal gradient and f_y the vertical gradient. $\mathbf{M}$ is called the structure matrix or nomal matrix. To detect feature points we know try to find points for which $\min \Delta^\top \mathbf{M} \Delta, ||\Delta|| = 1$ is large. This is the same as maximizing the eigenvalues of $\mathbf{M}$. The eigenvalue allow us to define a measure of "cornerness" (smaller k means more strict):

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

If we plot the values for the eigenvalues, we can divide the space as follows:
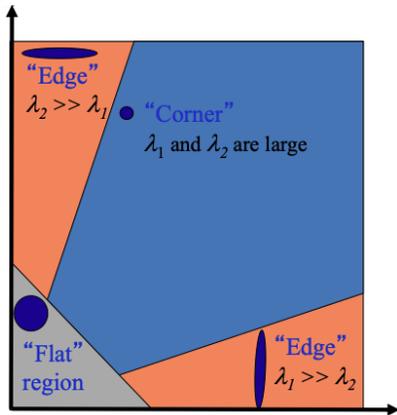
This is invariant to rotation since the eigenvalues remain the same after a rotation. But it is not invariant to scale, since the window is constant size. To make it scale invariant we can look for strong responses of DoG filter over scale space and only consider local maxima in both position and scale space.

To compare different images, e.g. for combining images. We use thresholded image gradients that are sampled over an array of locations in scale space. This can then be used for example to stich images together (SIFT).

6 Fourier Transformation

We have already seen the problem of aliasing. Now we want to understand how we can avoid aliasing and for this we introduce the **Fourier Transformation**.

The idea behind the Fourier transformation is to perform a change of basis, where the new basis elements are of the form $e^{-i2\pi(ux+vy)} = \cos(2\pi(ux+vy)) - i\sin(2\pi(ux+vy))$ (u, v are the parameters for the new basis).

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

The basis functions of Fourier transform are eigenfunctions of linear systems (one of the reasons it is so popular). Discrete Fourier transformation (DFT) can be represented as:

$$F = \mathbf{U}f$$

Here $\mathbf{U}$ is the Fourier transform base. The vector (u,v) determines the frequency by its magnitude and the orientation by its direction (only looking at the real part):
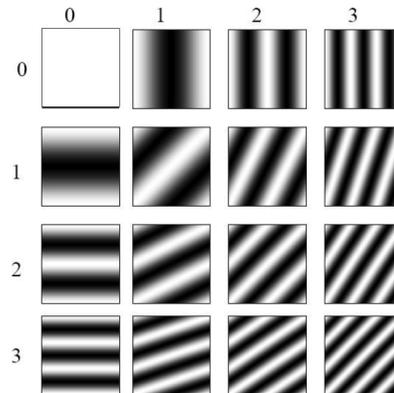

6.1 Phase and Magnitude

Since the Fourier transform is complex, it is difficult to plot. Instead we think of the phase (angle) and magnitude (length of the vector) of the transform. Note that natural images have about the same magnitude transform, hence phase seems to matter more. The phase part seems more random than the magnitude. Here you can see an example of the magnitude of an image.

6.2 Properties of the Fourier Transform

We already said that the Fourier transform is linear. Further it is important to note that the FT of a Gaussian is a Gaussian.
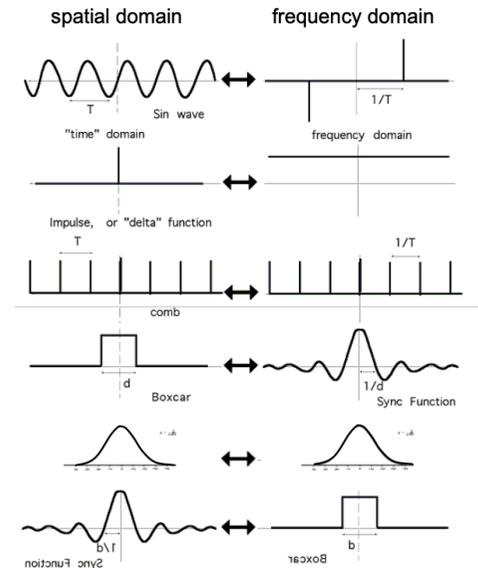
Convolution Theorem

The FT of the convolution of two functions is the product of their FT and the other way around:

$$F \cdot G = \mathbf{U}(f * g), \qquad F * G = \mathbf{U}(f \cdot g)$$

6.3 Sampling

Now we have a look at aliasing again. We define our sampling function as follows:

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

Where δ is the Dirac delta function. The FT of a sampled signal now corresponds to:

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

This approach can still lead to aliasing as high frequencies can lead to trouble. To avoid this we first need to suppress high frequencies before sampling. We can do this by convolution with a low-pass filter, this corresponds to multiplying the FT with the same filter. As a low-pass filter we use a Gaussian.

Nyquist Sampling Theorem

To understand why we need to do the low-pass filtering, we can take a look at the Nyquist sampling theorem: *The sampling frequency must be at least twice the highest frequency (of the signal).*

6.4 Signal Reconstruction

In image reconstruction we want to recreate our image from the samples data. To avoid pixelation we look at different reconstruction filters.
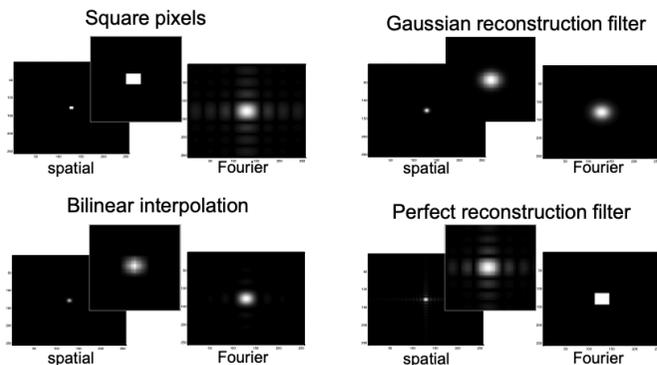
In the Fourier domain, determining the inverse of a kernel / filter becomes a lot easier, as:

$$F(h)(u,v) \cdot F(h^{-1})(u,v) = 1$$

With this we can for example try to remove motion blur from images. But we have to be careful to regularize our reconstruction filter to avoid noise amplification.

7 Unitary Transforms

Images can be either written as matrices or as vectors to make math easier. So a linear image processing system can be defined as:

$$g = \mathbf{A}f$$

So we ask ourself the question how to choose $\mathbf{A}$. We say $\mathbf{A}$ is unitary (or orthonormal for real values) if $\mathbf{A}^{-1} = \mathbf{A}^{H}$. Transformations by unitary matrices are energy conserving, meaning the length of the transformed vector will stay the same.

We introduce the following notation: f_i one image, $F = [f_1, ..., f_n]$ collection of images and $R_{ff} = E[f_i \cdot f_i^{H}] = \frac{F \cdot F^H}{n}$ image collection auto-correlation function. While unitary transformations preserve the energy, it will often be unevenly distributed among coefficients. The autocorrelation matrix of the transformed image will look like this:

$$R_{cc} = \mathbf{A} R_{ff} \mathbf{A}^{H}$$

The eigenmatrix Φ of R_{ff} is unitary and defined as follows:

$$R_{ff}\Phi = \Phi\Lambda \quad \text{— } \Lambda \text{ is a diagonal matrix of eigenvalues } \lambda_i$$

$$\Lambda = \begin{pmatrix} \lambda_0 & & & 0 \\ & \lambda_1 & & \\ & & \ddots & \\ 0 & & & \lambda_{MN-1} \end{pmatrix}$$
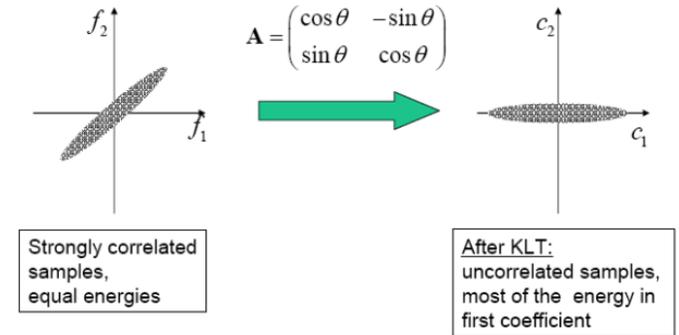
7.1 Karhunen-Loeve Transform / PCA

If we choose $\mathbf{A} = \Phi^{H}$ we get:

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

We can interpret this transformation as follows: "No other unitary transformation packs as much energy into the first

k coefficients, where k is arbitrary". The mean squared approximation error by choosing only the first k coefficients is minimized.

The basis images, which are the eigenvectors of the auto-correlation matrix, are called **eigenimages**. We can use eigenimages for recognition tasks, as the high dimensionality of the images space can be reduced to k dimensions. To perform recogintion, we tailor a KLT / PCA to the specific set of images we want to recognize, for example this lead to Eigenfaces.

The first principal component is the eigenvector with the largest eigenvalue. The eigenvalue can be interpreted as denoting the variance in the direction of the corresponding eigenvector. So the principal component shows in which direction the data is most spread out.

We can summaries the steps of PCA by:

- Subtract mean from the data and normalize
- Store data (images) as columns in a matrix
- Compute the covariance matrix of the resulting matrix
- Perform Eigenvalue Decomposition to get the eigenvalues and eigenvectors
- Sort the eigenvectors by descending value of corresponding eigenvalues
- Take the first k as principal component matrix U_k
- Project any further samples I into this space by multiplying the sample with the PC-matrix $U_k^{\top}(I - \text{mean})$

To restore a compressed image I_c we compute $I = U_k I_c + \text{mean}$.

7.1.1 Fisherfaces

To improve on the short comings of Eigenfaces, Fisherfaces try to find the direction where the ratio between individual variance are maximized. As the math behind this is rather complex, it is left out.
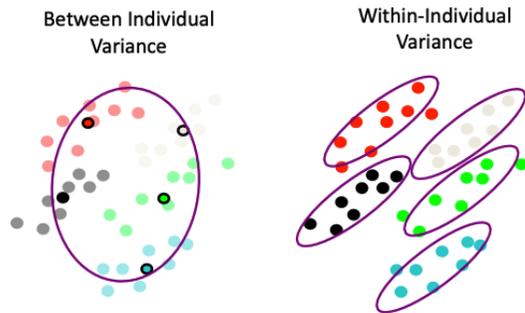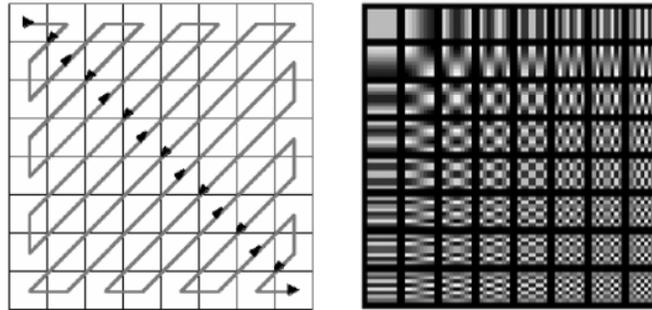

7.2 JPEG Compression

We notice that humans do not resolve high frequencies too well, therefore we can leave some of them away to reduce the size of an image.
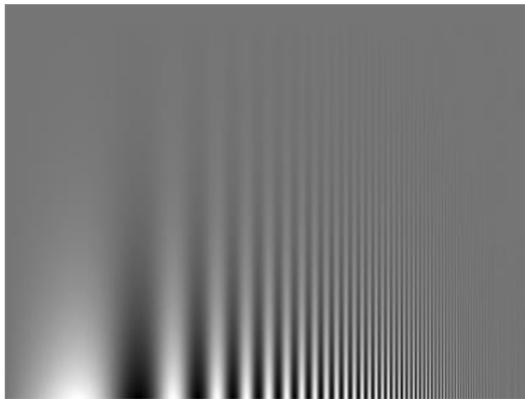
This is one of the main concepts in JPEG compression. Instead of FT, JPEG uses discrete cosine transform (DCT), which has no imaginary part. We go through the components of the DCT in a snake like pattern.


If the coefficients get too small after a certain point, we simply leave them away. In the end we apply Huffman encoding to further reduce the size of our image.

JPEG tends to introduce three kinds of distortions:

- General loss of sharpness and oscillations around high-contrast edges: these are due to approximating intensity transitions with smooth functions (cosines).

- Blocking structure: image is processed separately for every 8x8 block, block edges become visible at high compression ratios.

- Loss of color detail: the program may aggressively "downsample" chromaticity channels.

8 Pyramids and Wavelets

The **scale-space** is the family of signals generated by successive smoothing with a Gaussian filter. Using the scale-space we can downsample the image step by step, giving us an **image pyramid**.
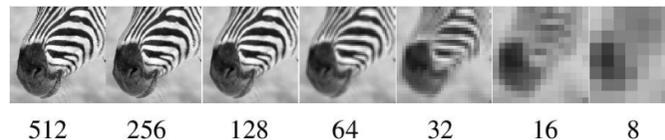

Such image pyramids can be used for edge tracking, search for correspondence, etc. One of their main benefits is that they allow for control of detail and computational costs.

8.1 The Laplacian Pyramid

While the Gaussian pyramid successively suppressed high frequencies, the Laplacian pyramid represents a different frequency at each level. This is similar to a bandpass filter.
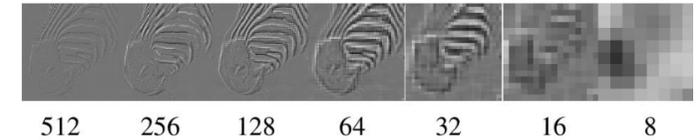
With a Laplacian pyramid we mean a difference of Gaussians (DoG). The main benefit is that it removes the redundancy of having the lower frequency parts in every level.

8.2 Wavelet Transform

The **Haar transform** is one of the most basic wavelets.
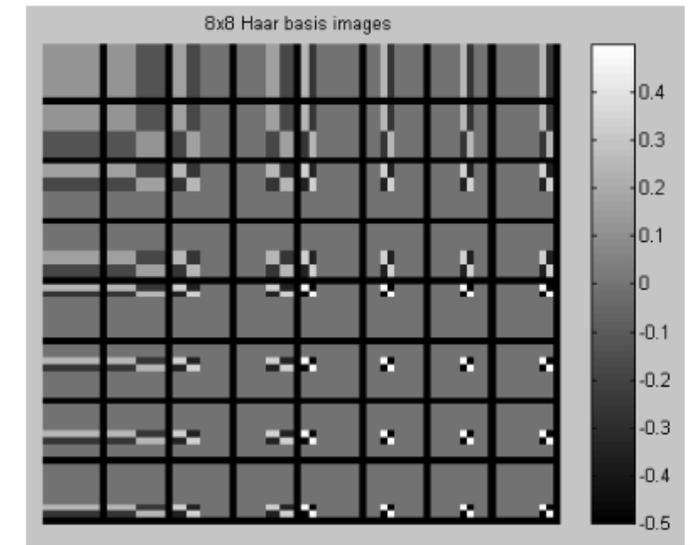
We can see that the outer vectors correspond to recursively applying a two-band filter to the bands of the previous stage (like a pyramid). The Haar transformation has poor energy compaction, meaning it is not really good for image compression.

In general wavelet transform works by splitting the signal into a low frequency and a high frequency pass, then the process is applied to the low frequency band recursively.

Analysis
Decomposition
DWT

Wavelet
Coefficients

Synthesis
Reconstruction
IDWT
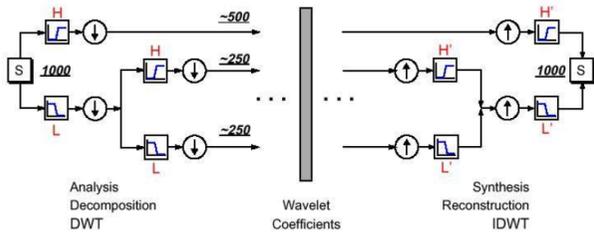
This concept is then again expanded to 2D images.

9 Optical Flow

The goal of optical flow is to estimate motion in videos. It is used for tracking, motion segmentation, video stabilization, compression, etc. Optical flow is defined as apparent motion of brightness patterns. This definition is important, as uniform, moving objects have an optical flow of zero, while non moving objects with change in lighting can have an optical flow not equal to zero.

We define $I(x, y, t)$ as the brightness of at (x, y) at time t. So the optical flow constraint is given by the equation:

$$\frac{\mathrm{d}I}{\mathrm{d}t} = \frac{\partial I}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial I}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{\partial I}{\partial t} = I_x u + I_y v + I_t = 0$$

9.1 Aperture Problem

If we look at the previous equation, we see that we have two unknowns u and v, meaning we have an under constraint problem. This is called the **aperture problem**. The aperture problem refers to the fact that when flow is computed for a point that lies along a linear feature, it is not possible to determine the exact location of the corresponding point in the second image. Thus, it is only possible to determine the flow that is normal to the linear feature.

The simplest solution to this problem is called the **normal flow**.

$$u_\perp = -\frac{I_t}{|\nabla I|}\frac{\nabla I}{|\nabla I|}$$

9.2 Regularization

Regularization introduces an additional smoothness constraint:

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

Beside the optical flow constraint:

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

Now we try to minimize $e_s + \lambda e_c$. This can lead to errors at boundaries as it is the opposite of what we try to enforce with the smoothness term.

9.3 Lucas-Kanade

We introduce the assumption of a single velocity for all pixels within an image patch:

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

We now have the constraints:

$$\frac{\mathrm{d}E(u,v)}{\mathrm{u}} = 0 \qquad \frac{\mathrm{d}E(u,v)}{\mathrm{v}} = 0$$

We solve with:

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

This is equivalent to:

$$\mathbf{MU} = \left(\sum \nabla I \nabla I^\top\right)\mathbf{U} = -\sum \nabla I I_t = b$$

The Lucas-Kanade algorithm works by computing $\mathbf{U}$ for at each pixel and then solving $\mathbf{MU} = b$. $\mathbf{M}$ is singular if all gradients point in the same direction, i.e. only normal flow is available.

We can refine our estimates by repeating the process multiple times and warping one image to the other using the estimated flow of the previous step.

9.4 Coarse to Fine

There are some failure modes to the local gradient method, for one if the intensity structure within our window is poor (uniform area). Another failure mode is when the displacement is too large or the brightness is not constant or if we have no spatial coherence. To make our approach more robust, we can again use an image pyramid, first estimating the optical flow on a coarse image and then iteratively start using finer images from the pyramid.

9.5 Parametric Motion Models

Global motion models offer more constraint solutions and integration over larger areas. For affine motions (rotation, translation, sheer) we introduce the following model:

$$I_x(a_1 + a_2 x + a_3 y) + I_y(a_4 + a_5 x + a_6 y) + I_t = 0$$

As each pixel provides one constraint in six global unknowns, we need a minimum of six pixels. The error we try to minimize here is again the square loss.

We can change the model further to allow for more types of transformation / warping of the image. There are also model that allow for 3D motion.

9.6 SSD Tracking

For large displacements we can use template matching, we do this by defining a small area around the pixel as the template and match the next image against that template. This will not work for uniform or noisy patches.

10 Video Compression

One of the main concepts of video compression is that while the human visual system is specifically sensitive to motion, some distortions are not as perceivable as in static images. Visual perception is limited to < 24Hz, but flicker can be perceived up to > 60Hz.

Bloch's Law

Up to a time frame of 100ms, it does not matter "how" light arrives only the sum, e.g. 10ms of double the intensity is equal to 20ms at the normal intensity.

10.1 Video Format

A video sequence is a bunch of images aligned in a time sequence. The interlaced video format uses two temporal shifted half images and increases the frequency from 25Hz to 50Hz.


top field   bottom field
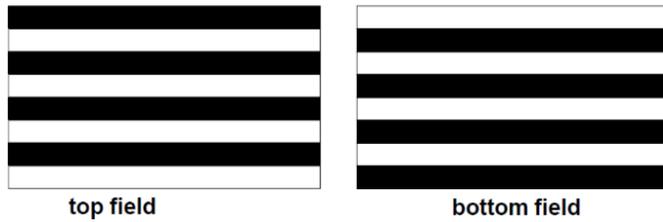
Today this is not done anymore, we rather use a progressive format that updates the whole screen at a time.

10.2 Temporal Redundancy

One way of compressing video is to take advantage of similarity between successive frames. Especially for high frame rates this works well.

The most used representation along the temporal dimension are predictive methods. There we define three typed of frames:

- **I Frame**: Intra-coded frame, independent of all other frames
- **P Frame**: Predictively-coded frame, based on the previous I and P frame
- **B Frame**: Bi-directionally predicted frame, based on both the previous and future I and P frames

I frame   P-frame   B-frame
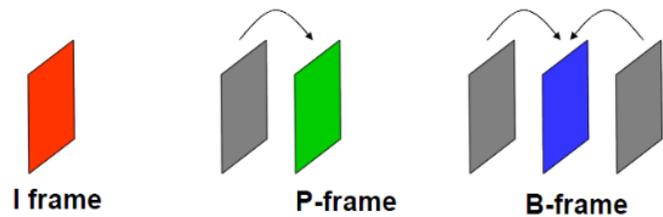
P frames can send motion vector plus changes. The frames starting at an I frame until the next I frame is also called a group of pictures (GOP). Temporal redundancy becomes inefficient if there are many scene changes or there is a lot of motion, e.g. confetti.

10.2.1 Motion-Compensation Prediction

One possibility of dealing with a high amount of motion is to use MC-prediction. Ideally we would partition the video into moving objects and describe these object motions. In reality this is very difficult, therefore we partition each frame into blocks, e.g. 16x16 pixels, and try to describe the motion of each block.

Motion Vector (mv_1, mv_2)

Reference Frame   Current Frame
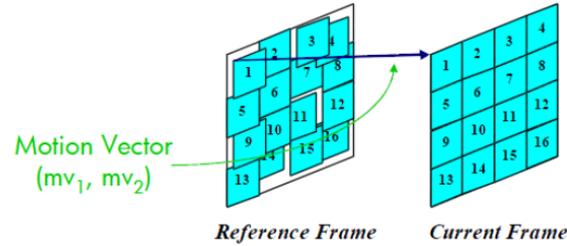
The algorithm first divides the frames into blocks and then uses the best matching blocks of the reference frame as prediction of the blocks in the current frame. For the block matching we use some best match metric, e.g. mean squared error or mean average error. The candidate blocks for the best match can either be determined by looking at all blocks or select a subset by some assumptions.

If we combine all the motion vectors, we end up with a **motion field** for all the blocks. As motion is not limited to integer-pixel offsets, one could try to estimate sub-pixel motion by spatial interpolation of the frames. The model of MC-prediction does not work well for more complex motions and might produce blocking artifacts.

### 10.2.2 Bidirectional MC Prediction

Instead of only looking at the previous frame we also take a look at the next frame. Bidirectional MC-prediction then estimates the position of a block in the current frame from either the previous frame, the next frame or the average of the previous and next frame.

10.3 Video Compression Architecture

Basic video compression architectures use temporal, spatial and color redundancies. Spatial redundancy uses DCT on the blocks and color redundancy does a color space conversion. A basic encoder could look as follows:
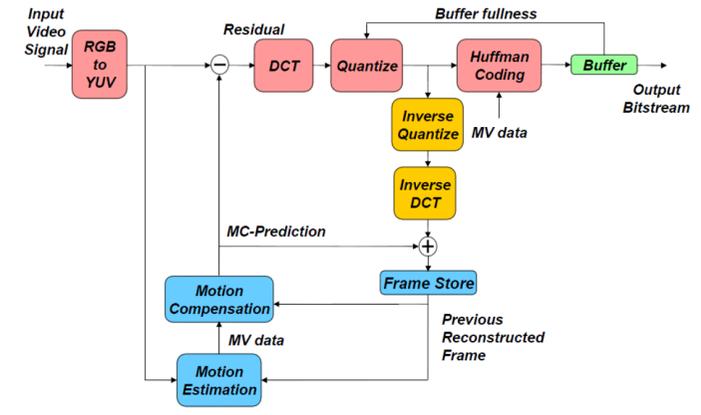

The corresponding decoder then looks like this:
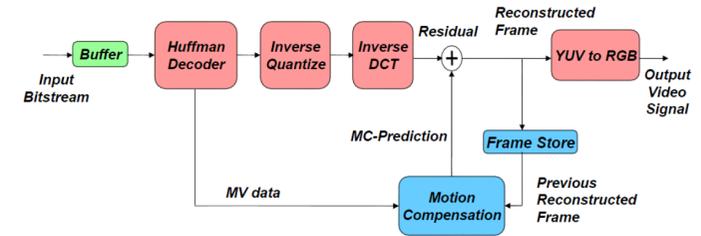
11 Radon Transform

Radon transformation is often used in medical imaging, e.g. computed tomography (CT). In CT data collection works by shooting x-rays through the material we try to image and collect them on the other side. If we do this from multiple angles, we can try to reconstruct an image from the measured data, as different material absorb a different amount of x-rays. We take the logarithm of this value, since absorption is a multiplicative process.

This can be seen as an image reconstruction problem.

$A + B = 7$
$A + C = 6$
$A + D = 5$
$B + C = 9$
$B + D = 8$
$C + D = 7$

problem   method   solution
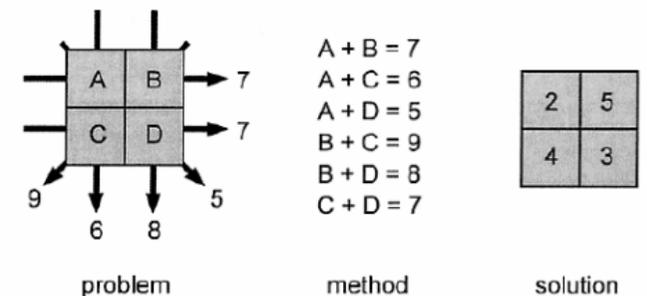
X-rays move along a straight line, at distance s it has intensity $l(s)$ and after traveling δs the intensity is reduced by δl. The reduction depends on the intensity and the optical density $u(s)$ of the material. For small δs it holds $\delta l / l(s) = -u(s)\delta s$. This leads to the following equations:

$$I_{\text{finish}} = I_{\text{start}} \cdot e^{R} \qquad R = \int_{L} u(s)ds$$

This related to the Radon transform:

$$Rf(L) = \int_{L} f(x)|dx|$$

Given the following setup:
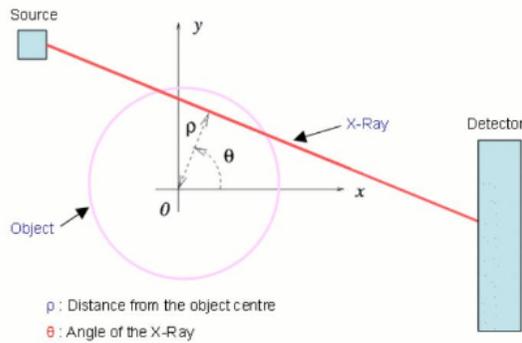
We can calculate:

$$R(p,\theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} u(x,y)\delta(p - x\cos\theta - y\sin\theta)dxdy$$

The Radon transform has the following properties:

- Linearity
- Shifting only changes the p coordinate
- Rotation of the coordinate system also rotates the Radon transformation
- The Radon transform of a 2D convolution is a 1D convolution of the Radon transformed function with respect to p

The following image shows the Radon transform of a square. We call this a **sinogram**.

f $\qquad$ $\mathcal{R}f$
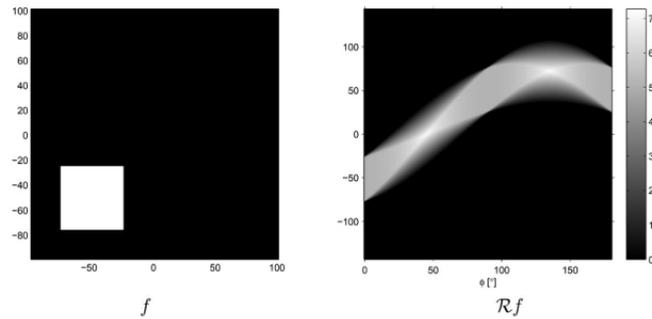
11.1 Backtransformation

Up until know we have seen how to perform a Radon transformation from an image to a sinogram. In reality we want the opposite, as medical imaging devices give us a sinogram and we want to reconstruct the image. This is the same as the question: "Can we find $u(x,y)$ if we know $R(p,\theta)$.

We can compute this with linear algebra, solving the overdetermined system $Kf = g$ using normal equations. This solution is not cheap, therefore we are interested in alternative ways. One possible way of doing this would be backprojection:
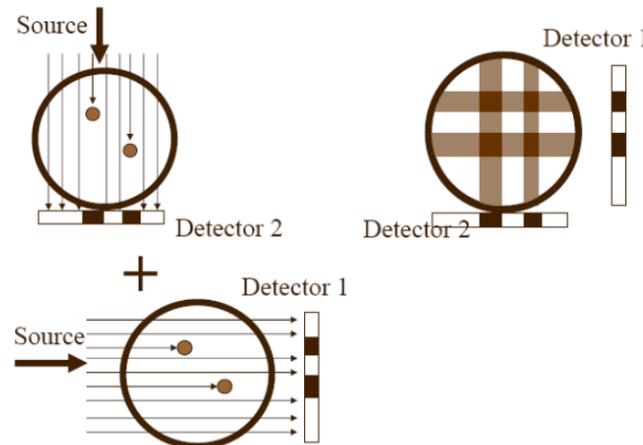
If we do this multiple times, adding more projections, we end up with a blurred version of the original image.

11.1.1 Fourier / Central Slice Theorem

$$G(q,0) = F(q\cos 0, q\sin 0)$$

This tells us that the 1D Fourier transformation of the measurement $g = Rf$ (for a fixed θ) is equal to the 2D Fourier transformation of the object slice $f(x,y)$ evaluated at a particular point. We can apply this for any orientation θ.

11.1.2 Backprojection Algorithm

The backprojection algorithm works as follows, for each of the K projection angles θ:

1. Measure projection data $P_{\theta}(t)$
2. Fourier transform it to find $S_{\theta}(w)$
3. Multiply by weighting function $2\pi|w|/K$ (high-pass filter)
4. Sum over the image plane and perform 2D inverse Fourier transform.
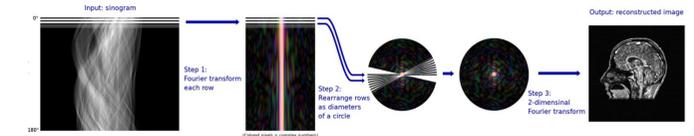
There are some practical issues, as it requires many precise measurements and is sensitive to noise it may lead to blurring in the final image.

12 Sparsity

We can use sparsity for image restoration. The idea behind this is, that a sparse signal is good for representing structure but not for white Gaussian noise. For this we model an image as follows:

$$\underbrace{y}_{\text{measured image}} = \underbrace{x}_{\text{original image}} + \underbrace{w}_{\text{noise}}$$

Now we perform MAP (maximum a posteriori) estimation:

$$E(x) = \frac{1}{2}||y - x||_2^2 + \Pr(x)$$

Where Pr is a log prior. Some classical priors to use would be smoothness ($\lambda||\mathcal{L}x||_2^2$ or total variation $\lambda||\nabla x||_1^2$.

Let $\mathbf{D} = [d_1, ..., d_p]$ be a set of normalized basis vectors, we call it **dictonary**. $\mathbf{D}$ is adapted to x if it can represent it

with few basic vectors, meaning a sparse vector α exists, so that $x \approx \mathbf{D}\alpha$.

There are predefined dictionaries, but we can also try to learn one ourself. We use the following model:

$$\min_{\alpha} \frac{1}{2}||x - \mathbf{D}\alpha||_2^2 + \lambda\psi(\alpha)$$

Here ψ induces sparsity and can be the $l_0, l_1, ...$ norm (we call it Lasso if l_1 is used). This idea can be expanded to not only perform denoising, but also do inpainting and demosaicking.

13 Texture

The key issues for textures are analysis / segmentation, representing the texture, and synthesis, generating textures.

13.1 Representing Textures

Textures are made up of stylised subelements, repeated in meaningful ways. To represent a texture we can try to find these subelements and represent their statistics. Finding the subelements can be done by applying filters and looking at the magnitude of the response. Possible filters could be spots and oriented bars at a variety of different scales (image pyramids). We do this by building a Laplacian pyramid of the image of the texture and then apply a number of oriented filters to each level. This represents image information at a particular scale and orientation.

13.2 Texture Synthesis

There is a variety of approaches to texture synthesis. One might use a histogram to capture the intensity probability distribution, but this does not capture any spatial relations. To improve on that one might use a co-occurrence matrix, having the probability distributions for intensity pairs. Other approaches are image-base, trying to perform "cut and paste". This is done by assuming Markov property and computing $P(p|N(p))$ (N is the neighborhood of p).

Synthesizing a pixel — non-parametric sampling — Input image
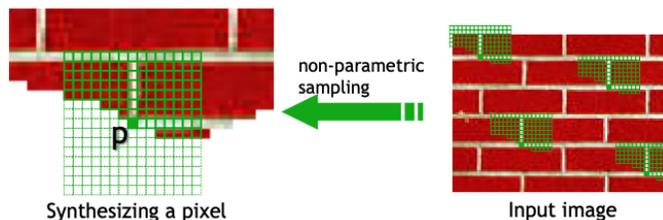
To synthesize p, just pick one match at random. This can be expanded to work on patches of images and not single pixels.

13.2.1 Chaos Mosaic

Another method for image synthesis is using a chaos mosaic. It works by tiling the input image, picking random patches and then smoothing the edges between them. This works best for random textures but will fail for structured ones.

Computer Graphics

14 Graphics Pipeline

The graphics pipeline contains all the steps we need to get from a geometric representation to a image. As an input we get not only the geometric representation (e.g. triangle mesh), but also information about materials, lighting and a virtual camera position.

The pipeline then consists of the following steps:

1. **Modeling Transform** - Take the object and transforms it into world coordinate space

2. **Viewing Transform** - Transforms the world coordinate space into camera coordinate space

3. **Primitive Processing** - Transforms the geometric representation into a triangle mesh

4. **3D Clipping** - Removes triangles not visible to the camera (objects outside the frustum)

5. **Projection to Screen Space** - Projects from 3D camera space to 2D screen space

6. **Scan Conversion** - Converts triangles to pixels and interpolates attributes

7. **Lighting, Shading, Texturing** - Computes color based lighting, shading and texture map

8. **Occlusion Handling** - Updates the color buffer using the depth buffer (z-buffer), deal with objects that are hidden by other objects

9. **Display** - Output to the display

If we perform the lighting, shading, texturing step after the scan conversion, we call it **pixel shading**, if it is the other way around, we call it **vertex shading**.

Programmer's View

From a programmers point of view the graphic pipeline looks as follows:


Vertex processing deals with per-vertex operations (vertex shaders) and fragment processing deal with per-pixel operations (fragment shaders).

15 Light and Colors

To understand colors in computer graphics, we first have to learn about the real physical properties of colors.

15.1 What is light?

Light is a form of electromagnetic radiation, we perceive a very limited section of the spectrum as visible light.
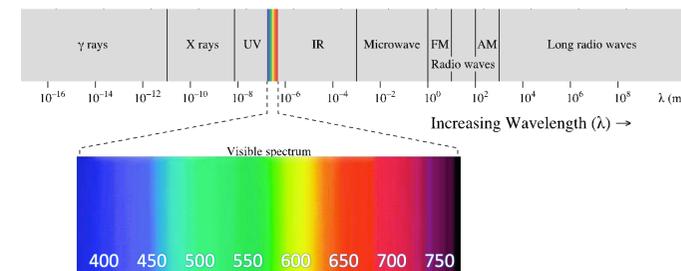
Light can be a mixture of many wavelength and is typically defined by the spectral power distribution (SPD). $P(\lambda)$ defines the intensity of this spectrum at wavelength λ. We humans then perceive this distribution as colors. A light

ray carries more information than a human can process, so we project this spectrum onto a 3D subspace given by the types of cones.

15.2 Anatomy of the Eye

On our retina there are two types of different cells, rods and cones. While rods respond to intensity only, the cones respond to color. There are three type of cones, each responding to a different wavelength (short = blue, medium = green, long = red). These cones are the reason we project the infinite dimensional space $P(\lambda)$ onto a 3D subspace.

15.3 Defining the Subspace

When doing this projection we will inherently lose some information, therefore two different SPD might look the same to us. There have been multiple attempts to standardise this subspace and the projection used.

The CIE Primary System

This was one of the the first attempts at standardising the color subspace. Their approach was to try to define every color by an additive mixture of the three base colors red, green, blue. They came up with the following results:
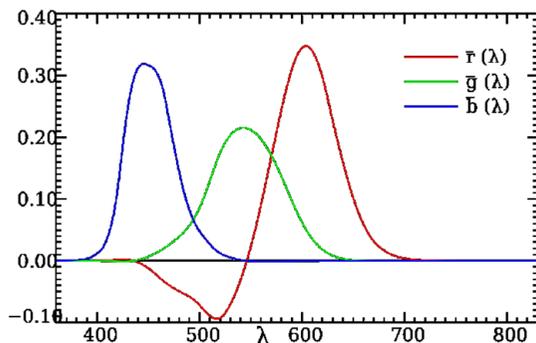
This shows that there are some colors that cannot be reproduced (negative r value). The CIE XYZ color space defines transformation from these three curves to a vector (X, Y, Z).
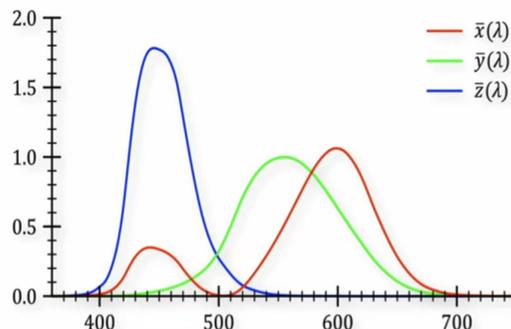
$$X = \int_0^\infty P(\lambda)\bar{x}(\lambda)d\lambda$$

$$Y = \int_0^\infty P(\lambda)\bar{y}(\lambda)d\lambda$$

$$Z = \int_0^\infty P(\lambda)\bar{z}(\lambda)d\lambda$$

The transformation matrix from RGB to XYZ is then given by:

$$\begin{pmatrix}\bar{x}(\lambda)\\\bar{y}(\lambda)\\\bar{z}(\lambda)\end{pmatrix} = \begin{pmatrix}2.36 & -0.515 & 0.005\\-0.89 & 1.426 & 0.014\\-0.46 & 0.088 & 1.009\end{pmatrix}\begin{pmatrix}\bar{r}(\lambda)\\\bar{g}(\lambda)\\\bar{b}(\lambda)\end{pmatrix}$$

These transformed curves have the properties that they are normalized, positive definite and the $\bar{y}(\lambda)$ curve corresponds to the luminance.
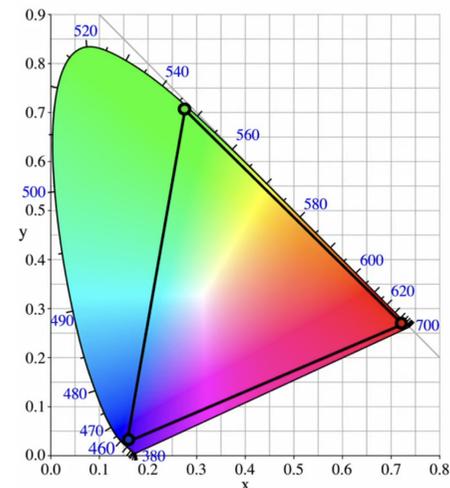

As the XYZ color space is 3D, it is often not very practical. Therefore we normalize the XYZ components and project it into a 2D space:

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z}$$

To reverse is given by:

$$X = x \cdot \frac{Y}{y} \qquad Z = \frac{Y}{y} - x \cdot \frac{Y}{y} - Y$$

Now (x, y) characterize color and Y characterizes brightness. If we plot (x, y) on a plane, we get all the colors of a single brightness (chromaticity chart).

The primary colors are on the boundary of the horse shoe and all linear combinations of two colors are on a line. The line between 380mm and 770mm is called the purple line and represents colors that are not spectral. Further if we have a point and draw a straight line connecting it with white and the pure color on the edge, we end up with a line sharing the same dominant wavelength. The position along this line characterizes saturation.

Screens are limited to a combination of three colors (triangle in the chart), so they can not show all possible colors.
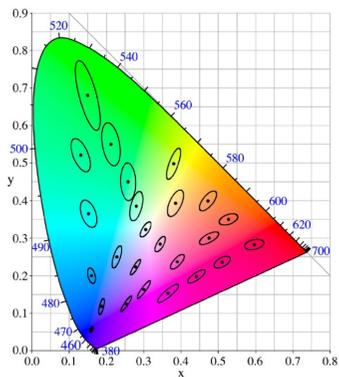
Other Color Spaces

The **CMY color space** is the inverse of the RGB space.

$$\begin{pmatrix}C\\M\\Y\end{pmatrix} = \begin{pmatrix}1\\1\\1\end{pmatrix} - \begin{pmatrix}R\\G\\B\end{pmatrix}$$

The main difference between RGB and CMY is that RGB is an additive color space, that is every color that can be represented is a non negative linear combination of the 3 primaries. CMY on the other hand is a subtractive color space, that is every color that can be represented in CMY is a non positive linear combination of the 3 primaries.

The **HSV color space** consists of hue (base color), saturation (purity of color) and value (brighness). This is a more user oriented color space, as it is rather intuitive to interact with.

MacAdams ellipses describe an area around a color, such that everything inside the ellipse is perceptual indistinguishable.

The **CIELAB and CIELUV color spaces** use MacAdams ellipses and transform the color space, so that the ellipses become nearly circular.
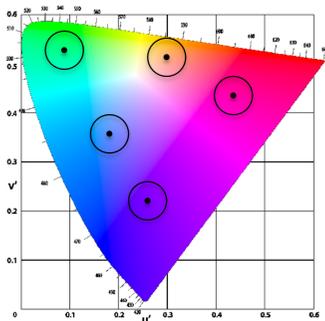
16 Transformations

Transformations map geometry. In the graphics pipeline this is mostly used for changing positions and orientations for objects, projecting them to the screen and animating objects.

16.1 2D Transformations

- Translation - note that translation is not linear as it corresponds to vector addition

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

- Scaling

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

- Rotation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Affine maps are linear in homogeneous coordinates (adding one dimension). This allows us to perform translations as a linear operation ($p' = \mathbf{T}p$).

- Translation

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Scaling

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Rotation

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Shear along x-axis (a) and along the y-axis (b)

$$\mathbf{SH} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{SH} = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When transforming back from the homogeneous coordinates it is important to divide by the last value. We can easily combine multiple transformation by taking the matrix product.

16.2 3D Transformations

We also want to use homogeneous coordinates in the 3D space.

- Translation

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Scaling

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation - this is **not commutative** anymore! For 3D rotations we have $R^{-1} = R^{\top}$.

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Shear parallel to the principal planes

$$\mathbf{SH}_{xy} = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{SH}_{xz} = \begin{bmatrix} 1 & sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{SH}_{yz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ sh_y & 1 & 0 & 0 \\ sh_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

16.3 Coordinate Systems

A coordinate system represents a point / vector as a linear combination of orthonormal basis vectors.

$$p = p_x x + p_y y + p_z z$$

A change of coordinate systems consists of a combination of translation and rotation:

$$p' = \mathbf{TR}p = \begin{bmatrix} r_1 & r_2 & r_3 & t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

We are also interested in how the surface normal n is transformed. Given a transformation $p' = \mathbf{M}p$ the surface normal is transformed by $n' = (\mathbf{M}^{-1})^\top n$.

16.4 Projection

When we want to go from 3D space to 2D space, e.g. when projecting to screen space, we have to think about perspective. We differentiate two types of projection.
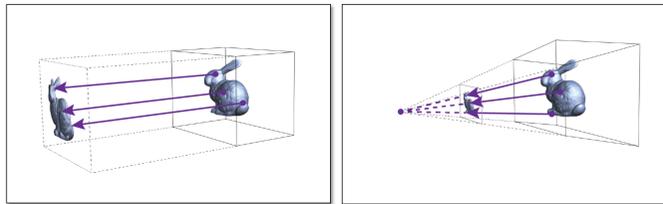
Parallel Projection   Perspective Projection

Perspective projection corresponds to how a camera would see an object. In perspective projection the lines seem to converge in some point, these points are called vanishing points. If we have more than three vanishing points we have multiple center of projection.

16.4.1 Mathematics of Perspective Projection

If the camera plane is defined by the x and y axis, i.e. the coordinate system is aligned with the z axis, the math behind the projection is easy. This is the reason we first transform from world coordinates to camera coordinates. For a given focal distance d, the projection is then given

by:

$$\mathbf{M}p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix}$$

If we transform this back to non homogeneous space, we end up with:

$$\begin{pmatrix} dx/z \\ dy/z \\ d \end{pmatrix}$$

16.4.2 Mathematics of Parallel Projection

This is a lot simpler and corresponds to:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

16.4.3 Projections in OpenGL

In OpenGL `modelviewMatrix` is used to project from object to camera space and `projectionMatrix` projects from camera to screen space.

17 Lighting and Shading

In lighting we have to differentiate between local and global illumination models. Local illumination models only consider the direct interaction of a light source with an object surface, the global illumination model also considers indirect lighting. We mostly focus on local illumination models.

17.1 Measuring Light

Radiometry is the study of measuring electromagnetic radiation, including visible light. We first introduce some definitions:

- **Angle** - $\theta = \frac{l}{r}$, for a circle 2π radians

- **Solid Angle** - $\Omega = \frac{A}{r^2}$, for a sphere 4π steradians

- **Direction** - point on the unit sphere parameterized by two angles $\omega = (\theta, \phi)$ (zenith and azimuth)
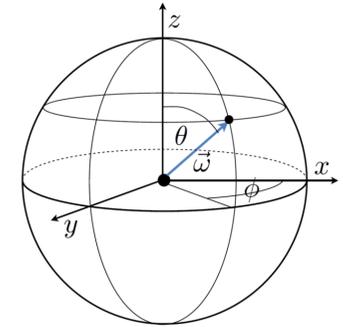
Now we can define light as consisting of photons with a position x, a direction ω and a wavelength λ. Each photon has an energy of hc/λ.

The basic quantities to measure light are then defines as:

- **Flux** Φ - total amount of energy passing through a surface or space per time unit

- **Irradiance** E - flux per unit area arriving at a surface

- **Radiosity** B - flux per unit area leaving a surface

- **Intensity** I - flux per solid angle

- **Radiance** L - intensity per unit area or flux density per unit solid angle

Flux		Irradiance	
$\Phi(A)$	$\left[\frac{J}{s} = W\right]$	$E(\mathbf{x}) = \frac{d\Phi(A)}{dA(\mathbf{x})}$	$\left[\frac{W}{m^2}\right]$
Radiosity		Intensity	
$B(\mathbf{x}) = \frac{d\Phi(A)}{dA(\mathbf{x})}$	$\left[\frac{W}{m^2}\right]$	$I(\vec{\omega}) = \frac{d\Phi}{d\vec{\omega}}$	$\left[\frac{W}{sr}\right]$
Radiance			
$L(\mathbf{x}, \vec{\omega}) = \frac{d^2\Phi(A)}{\cos\theta dA(\mathbf{x})d\vec{\omega}}$	$\left[\frac{W}{m^2 sr}\right]$		

17.2 Reflection Models

17.2.1 BRDF

Bidirectional Reflectance Distribution Function (BRDF) models the surface and its reflection of light. The BRDF provides a relation between incident radiance and differential reflected radiance.

$$f_r(x, \omega_i, \omega_r) = \frac{dL_r(x, \omega_r)}{L_i(x, \omega_i)\cos\theta_i d\omega_i}$$

From this we can derive the **reflection equation**:

$$L_r(x, \omega_r) = \int_{H^2} f_r(x, \omega_i, \omega_r) L_i(x, \omega_i) \cos\theta_i d\omega_i$$

The reflection equation describes a local illumination model. BRDF has the ability to express a large variety of complex materials. There are large libraries of measured BRDFs for different materials.

17.2.2 Simpler Reflections

These physical based models where for a long time to complex, so a simpler model was used. A material was characterized by a combination of diffuse and specular reflexion.

diffuse   specular   glossy
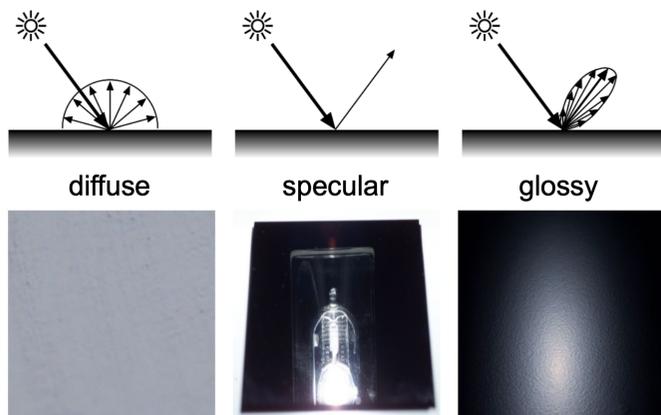
For diffuse reflection, the BRDF is a constant, since it is the same value over the whole hemisphere.

$$L_r(x) = f_r E_i(x)$$

17.2.3 Phong Illumination

OpenGL uses simplified reflections, one of the most used models is the phong illumination model.

Ambient Light comes from all directions. Its reflection is independent of the cameras position, light position and surface orientation. So we can model the reflection intensity by using a light source and a material parameter.

$$I = I_a k_a$$

Diffuse Reflection is dependent on directed light I_p (light source position) and the orientation of the source. It still

is independent of the camera position. So we can model the reflection intensity by either using a cosine or the dot product using the surface normal.

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

In a very simple model, we can now simply take the sum of the ambient light and the diffuse reflection.

$$I = I_a k_a + I_p k_d (N \cdot L)$$

To improve on this we can add quadratic attenuation due to spatial radiation (quadratically less energy when moving away from the source).

$$I = I_a k_a + f_{att} I_p k_d (N \cdot L) \qquad f_{att} = \frac{1}{d_L^2}$$

To further improve we can add the specular reflection. This depends on the angle between the reflection R and viewing ray V.
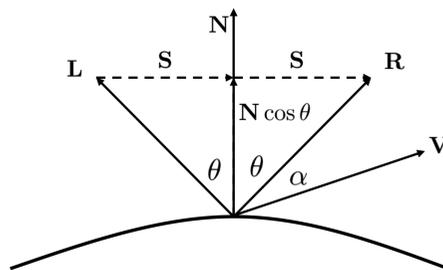
Combining this we end up with a simple but already quite good model for lighting.

Ambient   Diffuse   Specular
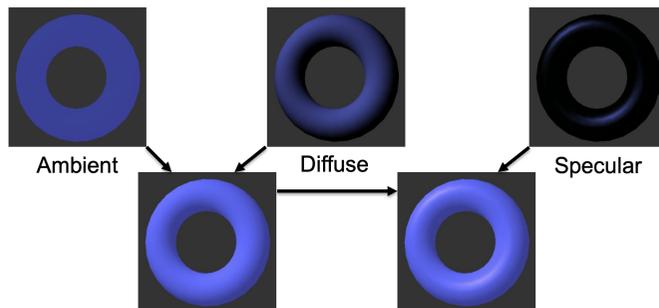
The phong illumination model approximates specular reflection by cosine powers. It also makes the whole function dependent on the wavelength.

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

17.3 Shading Models

We have seen how to calculate the lighting of an object, now we want to know how to calculate the color per primitive. The simplest model would be to have a single color per primitive, this is called flat shading and happens in screen space. This does not look good as a primitive converts to many pixels.

17.3.1 Gouraud Shading

Similar to flat shading, gouraud shading also happens in screen space. It works by calculating the face normals and then the vertex normals by averaging the face normals. After that we evaluate illumination for each vertex and then interpolate the vertex colors bilinearly on the current scan line.
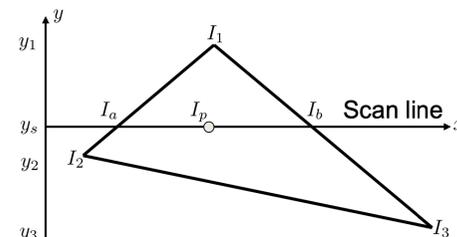
There are a few problems with scan line interpolation. The first one being perspective distortion, as we interpolate in the projection plane. Further the interpolation is orientation dependent and quality depends on the size of primitives.

17.4 Phong Shading

Phong shading happens in object space. It works by barycentric interpolation of the surface normals. The color is then determined by the interpolated normal.
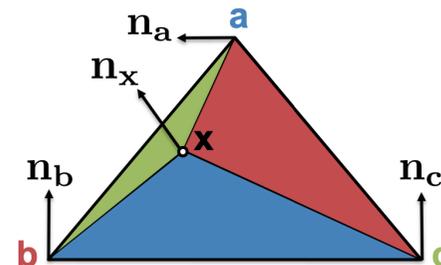
Phong shading cannot be applied if the normal is not defined.

17.5 Transparency

The typical color format includes an alpha channel, indicating the level of transparency. If we have partially transparent object, we need to blend the colors of the different objects together. One approach for this is **alpha blending**.
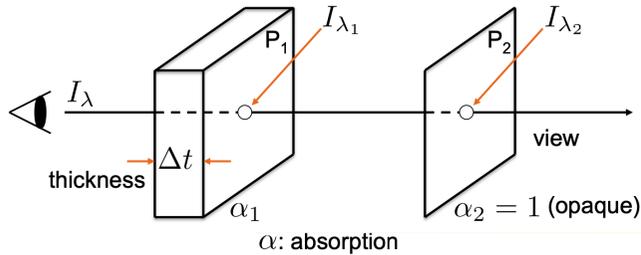
The intensity of the opaque object gets filtered by the object in front of it.

$$I_\lambda = I_{\lambda_1}\alpha_1\Delta t + I_{\lambda_2}e^{-\alpha_1\Delta t} \approx I_{\lambda_1}\alpha_1 + I_{\lambda_2}(1-\alpha_1)$$

To evaluate the alpha blending, we need to have an order to the primitives. This is the reason that we store the distance even in the screen space representation. Having that distance value (z-buffering) allows us to apply **back to front rendering** of the primitives. However this does not work anymore if we have intersecting objects.

To solve this problem we use **depth peeling**. It works by using multiple passes, each pass renders the next closest fragment.

18 Geometry and Textures

Geometry plays a fundamental role in CG. At the start, we need to define how we want to represent geometry in a discrete setting. Source data can be acquired by 3D scanning, digital modelling, procedural modelling, etc. We mainly differentiate by data generated on a computer (mesh) vs. data acquired from real-world object (point cloud).

18.1 Geometry Representation

If we look at geometry representation, we need to think about storage, acquisition / creating of shapes, editing of shapes and rendering of shapes.

Parametric Surfaces are surfaces that are defined by a parameter space. This allows us to simplify the storage to the position in the parameter space. On the downside, if there is no parameterisation of a surface, this does not work.

Subdivision Surfaces are surfaces that are piecewise linear. By increasing the subdivision, we can increase the number of surfaces and the precision.

Point Set Surfaces are a collection of points that can be combined to surfaces.

Polygonal Meshes store the boundary of objects and the connectivity.
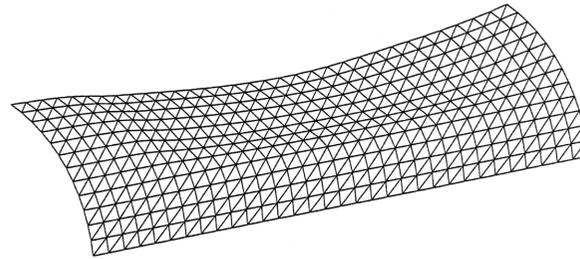

Todays GPU pipelines are optimized for such mesh structures and therefore provide fast rendering. On a mathematical level we define a polygon by a set of vertices $V = \{v_0, ..., v_{n-1}\}$ and a set of edges $E = \{(v_0, v_1), ..., (v_{n-2}, v_{n-1})\}$. A polygon is planar and non-self-intersecting.
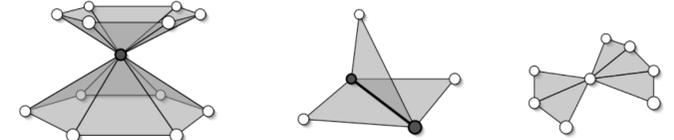
A polygonal mesh is a set of connected polygons $M = (V, E, F)$ where F are the faces of the polygons. It has the following properties:

- Every edge belongs to at least one polygon

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

A manifold is a surface locally homeomorphic to a disk.
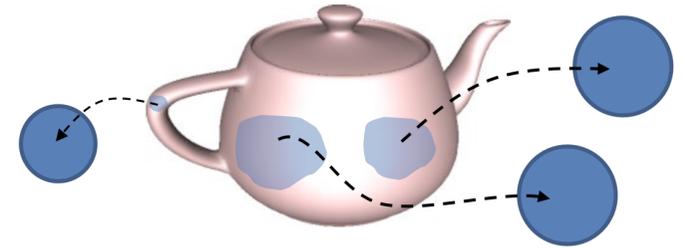
In a manifold mesh there are some structures that are not allowed:

Real-world data is often non-manifold.

18.2 Mesh Data Structures

We want to store the geometry and topology of a polygonal mesh. This data structure should allow for easy rendering, support geometry queries and allow for modifications.

Triangle Lists are a simple data structure. On the downside, they do not provide information about connectivity and are largely redundant (vertex gets saved multiple times).

| Triangles		
Vertex Coord.	Vertex Coord.	Vertex Coord.
(x_0, y_0, z_0)	(x_1, y_1, z_1)	(x_2, y_2, z_2)
(x_3, y_3, z_3)	(x_4, y_4, z_4)	(x_5, y_5, z_5)
...	...	...

Indexed Face Set are a lot more efficient, store connectivity and eliminate the redundancy. On the other hand geometric queries and modifications are costly.

Triangles		
Vertex Index	Vertex Index	Vertex Index
1	2	0
...	...	...

Vertices	
Index	Coord.
0	(x_0, y_0, z_0)
1	(x_1, y_1, z_1)
2	(x_2, y_2, z_2)
...	...

18.3 Texture Mapping

On fundamental way to enhance the quality of our renderings are texture mapping. Texture mappings increase the level of detail, without increasing our geometric mesh.
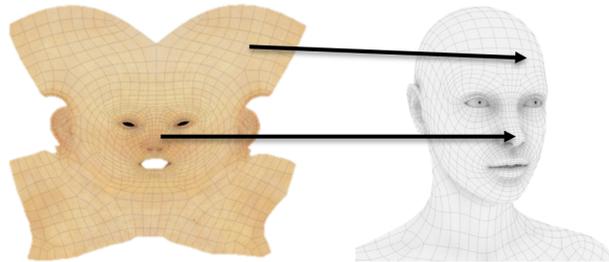
The difficulty is to find this mapping (2D to 3D space), this can lead to aliasing, blur or level-of-detail issues.

Texture mapping creates a one-to-one mapping between the texture and the geometry, this is done by parameterization of our geometric surface. We want the mapping to have the following properties:

- Low Distortion
- Bijective Mapping
- Efficient to Compute

Finding a good texture map can be done by using a texture atlas or by finding cuts to transform our 3D surface into 2D.

18.4 Texture Filtering

When we project our texture to a surface it is subject to aliasing.


Jaggy Edges　　Moiré patterns　　Loss of detail
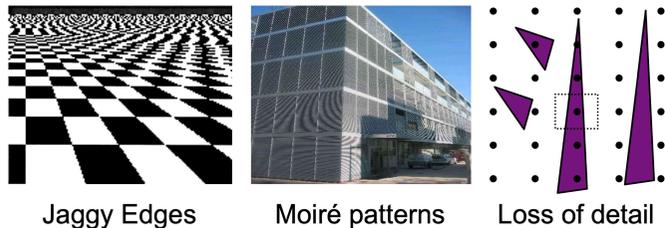
Low-pass filtering can help us avoid this aliasing. A typical low-pass filter would be the Gaussian filter.

Filtering in texture space is different form filtering in screen space. If we have an isotropic Gaussian in screen space this can relate to an anisotropic Gaussian in texture space.

Screen Space　　Texture Space
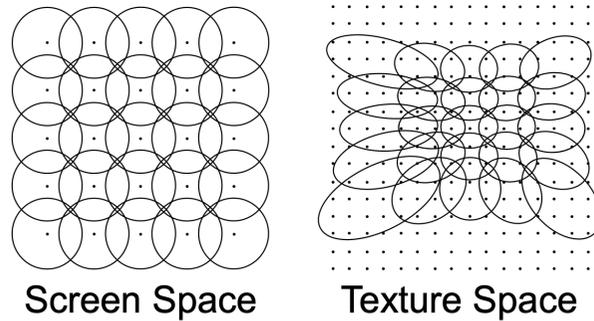
18.5 Light Map

Light maps are a trick to simulate the effect of a local light source.
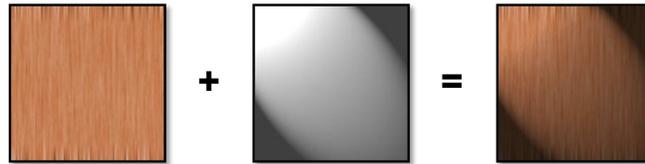
They allow for fast and high-quality shadows and lighting with an unlimited amount of light sources. On the downside they are not suited for every lighting model, take up space, do not work with moving light and take a lot of time to generate.
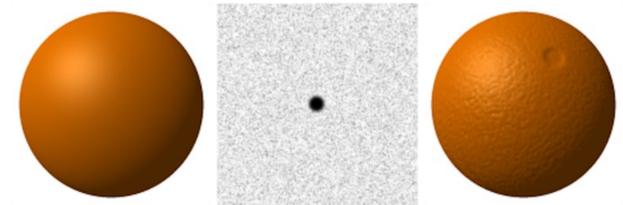
18.6 Environment Map

Environmental maps are used to render reflective objects efficiently. It works by by intersecting the reflected ray with the surrounding sphere or cube map.

18.7 Bump Mapping

Bump maps are used to perturb surface normal according to textures, allowing us to represent small-scale geometry (e.g. pores).


One limitation of bump mapping is that it does not work on the silhouette.

Normal maps are very similar to bump maps. The difference is that they contain a bit more information since they use RGB input and not only black and white.

18.8 Procedural Textures

Combining Perlin noise (or other types of noise) in different resolutions, allows us to create procedural textures. One examples for such a texture would be wood textures.
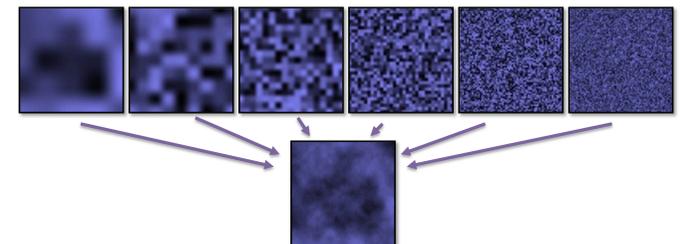
19 Processing Signals

This section focused on topic already covered in the first part of the lecture. Therefore I left out most of the content and only focused on the new content.

19.1 Antialiasing Filters

We already used Gaussian filters for antialiasing. A Gaussian Filter is a infinite response filter. The downside to this type of filter is that it is unstable. We mentioned that a Sinc Filter would be even better (ideal low-pass filter), but it is extremely hard to implement. B-Splines are another type of filter we really like, as they are easy to implement and allow for locally adaptive smoothing.

19.2 Perspective Projection

Equally distributed samples in texture space can get unequally distributed when projected into screen space. The optimal filter is spatially variant.

There are two type of problems that happen when projecting. **Magnification** happens when the pixel in the texture image maps to an area larger than one pixel. To avoid this we can simply use bilinear interpolation. **Minification** is the oposite, pixels in the texture image map to areas smaller than one pixel. To deal with this problem we use mipmapping.

Mipmapping is the process of storing the texture at multiple resolutions (image pyramid) and choosing the resolution level depending on the projected size of the triangle.

19.3 Geometric Aliasing

This is another problem that happens at the edges of polygons. Since we our screen space allows for only a limited number of pixels, we end up with a staircase pattern for the edges. To lessen this effect we can apply **supersampling**.

Supersampling introduces multiple color samples per pixel, which then are averaged. There are multiple possible sampling patterns, including uniform, jittering, stochastic and poisson. The following shows supersampling using jittering.
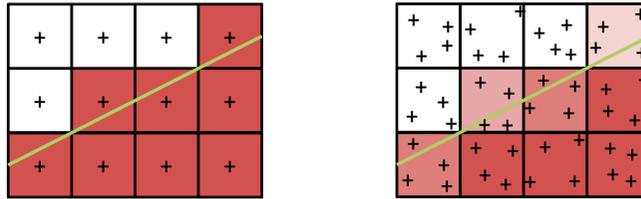
20 Bezier Curves

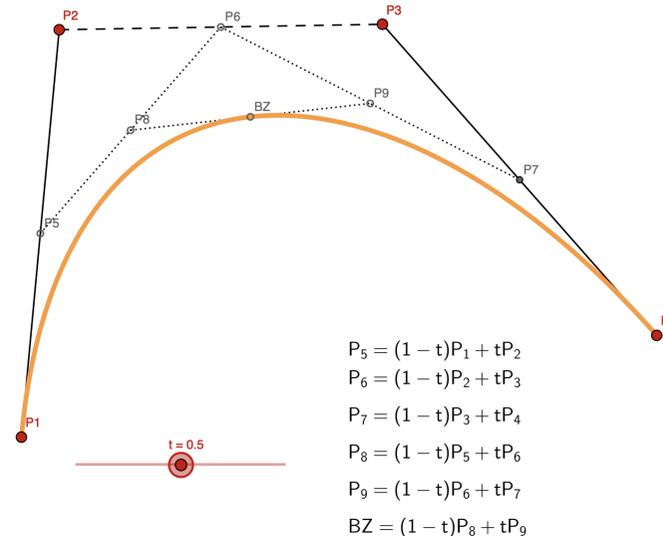I would highly recommend watching this YouTube video for a detailed explanation of the topic.

One way of modelling a geometric object are Bezier curves. Bezier curves are defines as:

$$x(t) = b_0 B_0^n(t) + ... + b_n B_n^n(t)$$

Where $B_i^n(t)$ is a Bernstein polynomial of degree n:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-1} \qquad i < 0, i > n : B_i^n(t) = 0$$

Bernstein polynomials are positive definite and provide global support. Bezier curves can be thought consecutive interpolation.

$$P_5 = (1-t)P_1 + tP_2$$
$$P_6 = (1-t)P_2 + tP_3$$
$$P_7 = (1-t)P_3 + tP_4$$
$$P_8 = (1-t)P_5 + tP_6$$
$$P_9 = (1-t)P_6 + tP_7$$
$$BZ = (1-t)P_8 + tP_9$$

This leads to deCasteljau's algorithm, $\mathcal{O}(n^2)$. One of the downsides when combining Bezier curves is that the are only C_0 smooth.

21 B-Splines

B-Splines are in many ways a improvement over Bezier curves. They give us control over the continuity and allow for local support only. Cubic B-Splines are the most popular type, but we can define them for arbitrary degree.

A B-Spline $s(u)$ is built from piecewise polynomial bases and a sequence of knots $u_0 < u_1 < ...$

$$s(u) = \sum_{i=0}^{k} d_i N_i^n(u)$$

They have the following properties:

- **Partition of Unity**: $\sum_i N_i^n(u) = 1$
- **Positivity**: $N_i^n(u) \geq 0$
- **Compact support**: $N_i^n(u) = 0$, $u \notin [u_i, u_{i+1}]$
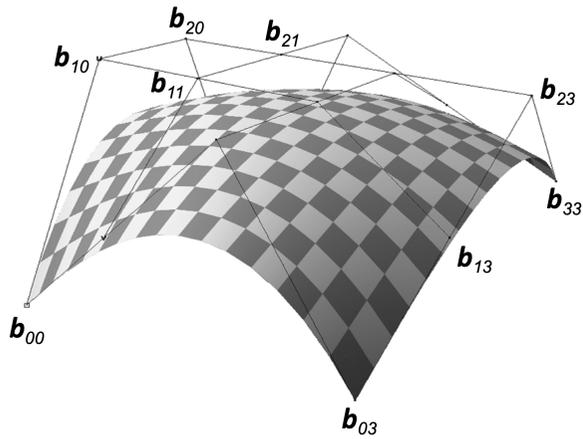- **Continuity**: N_i^n is (n-1) times continuously differentiable

The basis functions are defined as:

$$N_i^n(u) = (u - u_i)\frac{N_i^{n-1}(u)}{u_{i+n} - u_i} + (u_{i+n+1} - u)\frac{N_{i+1}^{n-1}(u)}{u_{i+n+1} - u_{i+1}}$$

### 21.1 Tensor Product Surfaces

Getting from a curve to a surface involves taking a tensor product (outer product). Given two Bezier curves $b^m(u)$ and $b^n(v)$ we can create the surface:

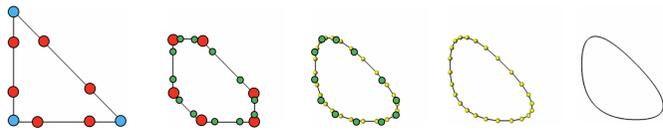$$b^{m,n}(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} b_{i,j} B_i^m(u) B_j^n(v)$$

One of these algorithms is the **Loop Subdivision**. It is a generalization of box splines and can be used on triangle meshes. It generates a G^2 continuous limit surface:

$$E_i = \frac{3}{8}(d_1 + d_i) + \frac{1}{8}(d_{i-1} + d_{i+1})$$

$$d_1' = \alpha_n d_1 + \frac{(1-\alpha_n)}{n}\sum_{j=2}^{n+1} d_j$$

$$\alpha_n = \frac{3}{8} + \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n}\right)^2$$

22 Subdivision Surfaces

Subdivision surfaces are a generalization of spline curves / surfaces. They allow for successive refinement and converge to a smooth surface.
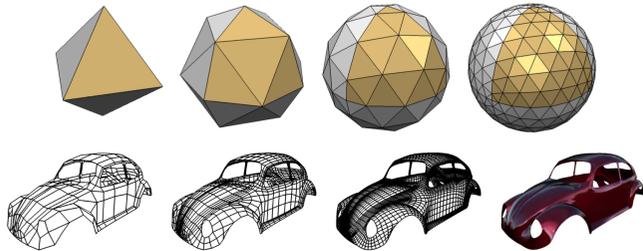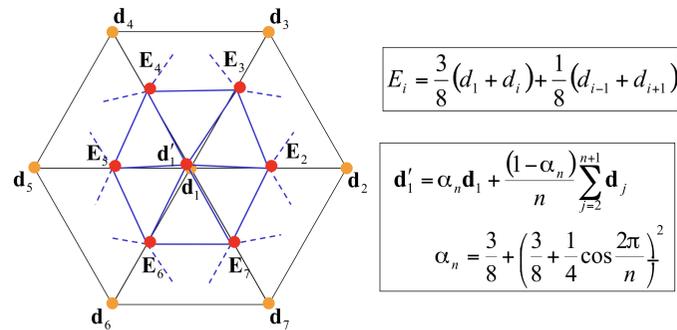

It relies on corner cutting:

1. Insert two new vertices at 1/4 and 3/4 of each edge

2. Remove the old vertices

3. Connect the new vertices

There are multiple algorithms for subdivision, with different properties (applied on polygonal mesh or triangle mesh, G^1 or G^2 continuous etc.).

23 Scan Conversion

Scan conversion (also called rasterization) is the problem of converting our objects to the discrete pixel space, e.g. deciding which pixel lie inside our object.

23.1 Scan Conversion of Lines

Bresenham lines choose the closest pixel at each intersection. The goal is to have a fast decision which pixel has to be drawn next. It does this by using the position of the midpoint m with respect to the intersection point q as a criterion.
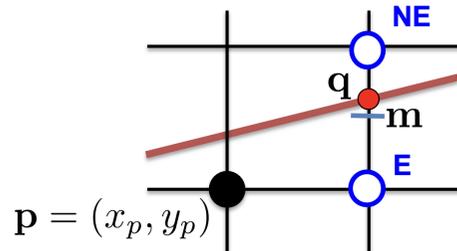
Given $f(x, y) = ax + by + c = 0$ as the implicit equation of the straight line, it decides based on $d = f(m)$. If $d < 0$ select pixel E, else select pixel NE. After the first decision, we can base our update criterion based on the choice E or NE.

$$\text{Pixel NE} : d_{new} = d_{old} + \Delta y - \Delta x$$

$$\text{Pixel E} : d_{new} = d_{old} + \Delta y$$

23.2 Scan Conversion of Polygons

Filled polygons (especially triangles) are the most important graphics primitives. The straightforward solution would be to perform an inside test for each pixel, but this is very inefficient. Instead we process scan line after scan line. We call a group of picked pixels inside a scan line a **span**. The algorithm then works as follows:

1. Calculate all intersections of the scan line

2. Sort the intersection points by ascending x-coordinates

3. Fill all spans in between two consecutive intersection points if the parity is odd
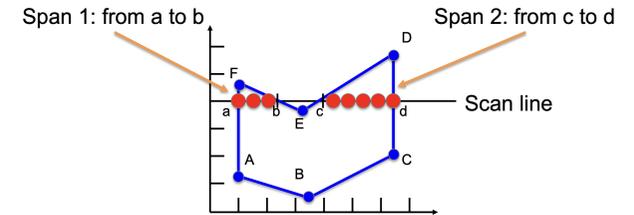
24 Visibility and Shadows

24.1 Visibility

We already encountered the **visibility problem**, some parts of some surfaces could be occluded. There are two possible solutions:

- **Painter's Algorithm** - Render objects / polygons from furthest to nearest. This leads to problems when there are intersections or cyclic overlaps.

- **Z-Buffering** - Store depth to the nearest object for each pixel.

The Z-Buffering algorithm works as follows:

1. Initialize all z values to ∞.

2. For each polygon: If z value of a pixel for this polygon is smaller than the stored z value, replace the stored z value.

The problem with this approach are that the resolution of the z-buffer is limited, we have to decide how many bits we need for the depth. If we think about how many bit to allocate, we also have to take into account that the depth resolution is not linear.

24.2 Shadows

Shadows are important as they can make an image immediately more realistic. There are different methods of creating shadows.

24.2.1 Basic Shadows

Planar shadows are the most basic approach, we simply draw a projection of the object on the ground. It is limited as it cannot create shadows on curved surfaces or other objects.
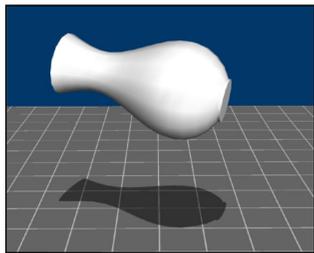
Projective texture shadows uses texture mapping in a projective way. It works by creating a black and white image of the object from light. Then it uses this image as a projective texture. It is again limited as we need to define the obstacle and the receiver and it does not allow self-shadows.

24.2.2 Shadow Maps

The fundamental idea of shadow maps it to compute the depths from the light source. The depth from the light source gets save to a z-buffer (light source coordinates) we call this a **shadow map**. If we then render the scene from the camera position, we can transform each pixel x_C from the camera coordinate system into the light coordinate system x_L. By comparing the depth of the corresponding value from the shadow map $d(x_L)$ and the projected point z_L, we can determine if a pixel gets illuminated or not.
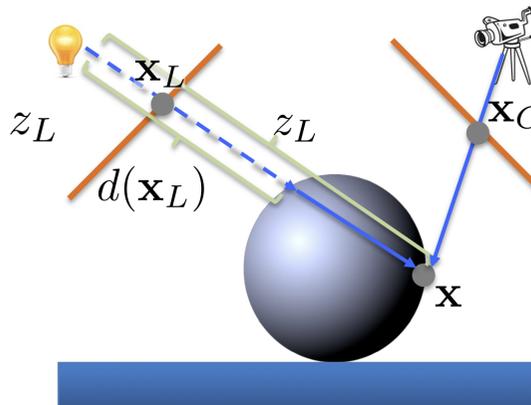
One of the main limitations of shadow maps is that there is a bias. Due to numerical errors it can happen that for a visible point $d(x_L) < z_L$. To avoid this we have to carefully select a bias to add to $d(x_L)$.

Another problem is the field of view. A point to shadow can be outside the field of view of the shadow map, due to the camera frustum. A solution to this would be to use a cubical shadow map or spot lights.

We might also occur aliasing, but we cannot filter depth. Instead we filter the result of the test by taking a weighted average of the comparisons.

24.2.3 Shadow Volumes

A third, more geometric, approach to rendering shadows are shadow volumes. For each light source we can explicitly represent the volume of space that is inside the shadow. If a polygon is inside the volume it is inside the shadow. To determine if a primitive is inside we apply the following algorithm:

1. Shoot a ray from the camera

2. Increment / decrement a counter each time the boundary of a shadow volume is intersected

3. If the counter is 0, the primitive is not in the shadow

This can be further optimized using silhouettes. Nevertheless this is very costly, as it introduces a lot of new geometry and we can only use the optimizations if objects are watertight.

25 Ray Tracing

The idea behind ray tracing the reverse process of what we did until now. We start by shooting rays through each pixel in our image.
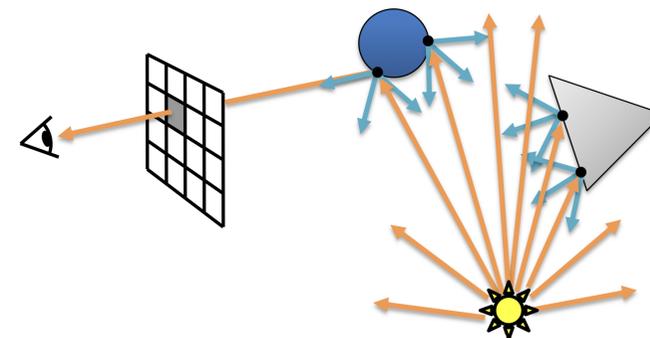

The first versions of ray tracing stoped at the first intersection with an object and then evaluated the illumination model for this point (also called **ray casting**). Nowadays **recursive ray tracing** calculates refractions of that initial ray until we arrive at the light source.

25.1 Forward Ray Tracing

Forward ray tracing would theoretically start at the light source and trace each ray until it hits the camera. This is not very efficient as most rays will never hit the camera.

25.2 Backward Ray Tracing

Backward ray tracing therefore does it the other way round. It traces rays from the camera into the scene until arriving at a light source. This easily solves some of the problems we previously encountered, e.g. hidden surface problem or transparency. When arriving at a object we calculate the lighting model for this point by casting additional rays depending on the type of surface (diffuse, glossy, specular). Additionally to determine shadows, we cast a ray towards the light source, if the ray arrives at the light source our point does not lie inside a shadow.
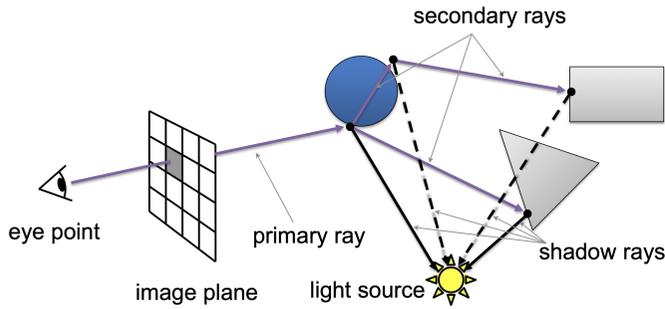
The basic pipeline therefor consists of the following steps:

- Ray Generation
- Intersection
- Shading

This three stages get repeated over and over. If we shoot multiple ray per pixel (supersampling) we not only improve the quality, but also implement anti-aliasing.

The ray equation is given by:

$$r(t) = o + td$$

o is the origin of the ray and d the direction. Using this we can calculate the intersection with various types of primitives, e.g. triangles or spheres.

25.3 Shading

We already said that physically correct shading is too expensive. Therefore we need to simplify by making some assumptions. We assume that there are different types of surface reflectance: diffuse, specular, ambient and transparency terms. Further we use the shadow ray to determine if a point is illuminated.

26 Acceleration Data Structures

There are three general techniques to speed up ray tracing. These are fewer intersection computations, fewer rays and generalized rays. We will focus on the first one, trying to calculate fewer intersections.

Ray-surface intersection is at the core of every ray tracing algorithm, the brute force approach computes the intersection of every ray with every primitive. That are many unnecessary ray-surface intersection tests.

Before we look at the actual acceleration data structures, we need to define the **axis aligned bounding box (AABB)**. This bounding box aligned with the three axis of our coordinate system and can be defined by two points min and max. We can then calculate if a ray intersects a bounding box by the following calculations:

The main focus of the acceleration data structures is to decompose space into disjoint regions and store pointers to overlapping objects within each region. When rendering we then have to traverse through regions overlapping the ray and calculate only the intersections for objects in each region until a hit is found.

26.1 Uniform Grids

The most basic of these data structures are uniform grids. We first need to compute the bounding box and then apply a determined grid resolution (often $\sim 3\sqrt[3]{n}$). Then we insert objects into cells, prune empty cells and store references for each object in a cell.
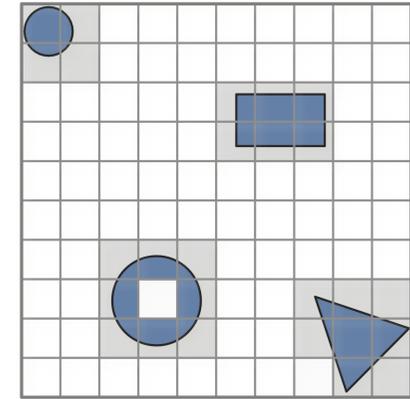

During rendering, we incrementally rasterize the ray and compute the intersection with objects in each cell until we found a intersection.

This already gives a huge improvement over the brute fore approach and can be further refined by using hierarchical grids.

26.2 Binary Space Partition

There are different type of spacial hierarchies. They all follow the same divide-and-conquer approach. This again results in a lot fewer intersection tests.

| octree | kd-tree | bsp-tree |
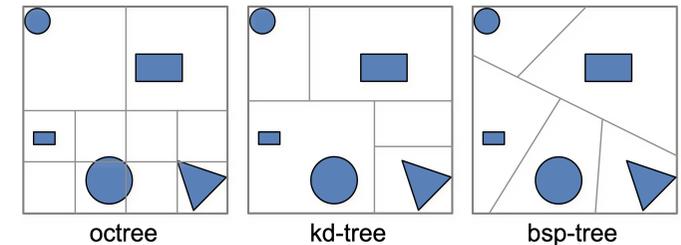
From left to right we have increasing complexity but also more flexibility and fewer intersection tests.

26.3 Bounding Volume Hierarchies

Bounding volume hierarchies are an alternative divide-and-conquer method. It works by decomposing object into sets (overlapping) and bound them using simple volumes (spheres, axis-aligned bounding boxes, oriented bounding boxes). We can either apply top down or bottom up construction for this data structure.
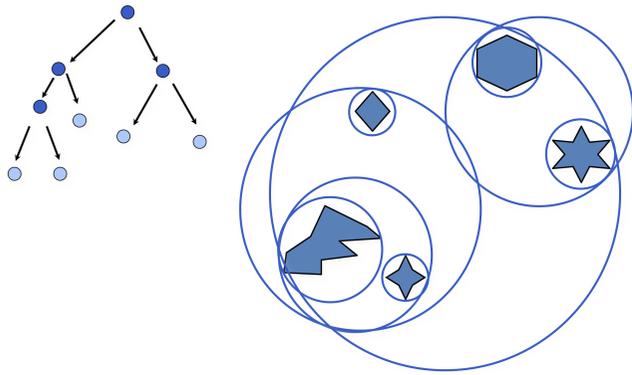
27 Animation Physics

Animation is about creating the illusion of motion, e.g. by rapidly displaying of a sequence of static images that minimally differ from each other. There are three basic techniques to animate motion in computers:

- Artist Directed, e.g. using keyframes

- Data Driven, e.g. motion capture

- Procedural, e.g. simulation

27.1 Key Framing

The idea behind key framing is to specify important events only, computer fills in the rest via interpolation or approximation. We also call this event and they do not have to be position, it could also be color, lighting, etc.

How do we interpolate data? Linear interpolation is simple but yields rather rough motion. A more appropriate interpolation technique would be to use spline interpolation.

27.2 Splines

In general, a spline is any piecewise polynomial function defined by data points (t_i, f_i), so that $f(t_i) = f_i$. The only other condition is that the function is a polynomial when restricted to any interval between knots.

Most commonly we use cubic splines (degree three). As cubic polynomials have four degrees of freedom, there are multiple solutions to interpolate given points. To find a solutions we need further restrictions, so we match the derivatives at the endpoints.

27.2.1 Natural Splines

Piecewise spline made up of cubic polynomials p_i.
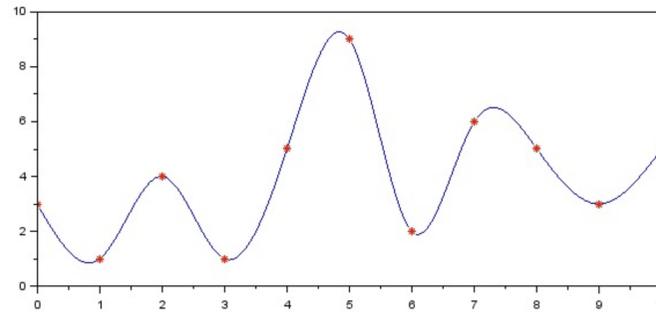
We want that these polynomials are C^2 continuous at the knots. This leaves us with two degrees of freedom remaining, so we have to introduce conditions for the endpoints. We set the curvature to zero at endpoints, therefore we can solve this system.

This function now interpolates the data and is C^2 continuous everywhere, but there is still a problem. If we move one knot, it will have an influence on the whole curve (locality).

27.2.2 Hermite / Bezier Splines

We have already seen B-Splines, they fulfil the locality condition but the C^2 continuity is gone. For each knot we have to additionally specify the tangents. These type of splines are the most used in computer graphics and animation.

27.2.3 Catmull-Rom Splines

Sometimes makes sense to specify tangents, but often it is more convenient to just specify values. Catmull-Rom splines are a specialization of Hermite splines, determined by values alone. They use difference of neighbors to define the tangents. Commonly they are used to interpolate motion in computer animation.