Visual Computing

Digital image

An image is a 2D signal, where by signal we mean a function depending on some variable with physical meaning (e.g. brightness, temperature, pressure, ...). We can represent images as functions $\mathbb{R}^2 \to \mathbb{R}^n$, where we give a pixel coordinate as input and we get a value as output. If we use a grayscale representation then we will get an value in $\mathbb{R}$, if we use RGB a value in $\mathbb{R}^3$.
A very important concept when we talk of visual computing is **sampling**. Sampling in 1D takes a function and returns a vector whose elements are values of that function at the sample points. For example if we do three samples of the function $\sin(x)$ we can get $\{(0,0),(\frac{\pi}{2},1),(\frac{3\pi}{4},\frac{\sqrt{2}}{2})\}$ as collection of vectors. With sampling we can represent continuous functions in a discrete manner, but it is often useful to have some reconstruction algorithm which, given some samples, approximates the continuous function. An application of this are CDs: when we record we sample the sound and we store it on the disc and in order to do the playback we need a device which takes the discrete samples on the CDs and returns the (continuous) sound. Of course, when we go from a discrete to a continuous world, one needs to guess what the function did in between. A famous way is to do bilinear interpolation, i.e. $f(x) = (1-a)(1-b)f(i,j) + a(1-b)f(i+1,j) + abf(i+1,j+1) + b(1-a)f(i,j+1)$. It is important to know how many samples to do. In facts there is a trade-off: doing too less samples implies a bad representation of the signal, doing too many samples is inefficient. Undersampling is dangerous because information is lost and we loose higher frequencies. An example of bad consequence of undersampling is aliasing: imagine that we take a picture of a watch every 55 seconds, one would have the feeling that the watch is going backwards.
Another important concept is **Nyquist-Shannon sampling theorem** which states: take the highest frequency of your signal, sample at least at the double of this frequency in order not to have an aliasing effect.
When it comes to digital images we also have a problem of **quantization**. In facts a signal is a real value, but this value gets approximated when stored in a digital devices (which uses a limited number of bits). Quantization is lossy, i.e. after quantization the original signal cannot be constructed anymore. This is in contrast to sampling, because sampled but not quantized signal can be reconstructed.

Segmentation

We take a first look to the problem of **segmentation**, the ultimate problem in computer vision. Segmentation partitions an image into regions of interest and is the first stage in many automatic image analysis systems. A first thing to keep in mind is that the quality of a segmentation depends on what you want to do with it, hence one always have to have an application in mind.
A simple segmentation process is thresholding, which labels each pixel as in or out of a region of interest by comparing the greylevel with a threshold T. Hence we produce a binary image B with the following rule:

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

In order to select a good value of the threshold we can either use trial and error and then comparing the result with groundtruth or use automatic methods such as ROC curves.
ROC curve: characterizes the error trade-off in binary classification tasks. We plot the TP fraction (sensitivity) and the FP function (1-specificity). In general any binary test has four possible outcomes:

True positive	False negative
True negative	False positive

We define the true positive fraction as:

$$\frac{\text{True positive counts}}{\text{Total number of positives}} = \frac{TP}{TP+FN}$$

And the false positive fraction as:

$$\frac{\text{False positive counts}}{\text{Total number of negatives}} = \frac{FP}{FP+TN}$$

In a ROC plot we try different thresholds and we draw the corresponding point in the plane. Every ROC curve always passes from $(0,0)$ and $(1,1)$ (this holds because by doing an extremely large/ small threshold we can classify something as always 1 or always 0) and the ROC of a perfect system touches the point $(1,0)$. But how do we choose the threshold? We choose an operating point by assigning relative costs and values to each outcome. Then we choose point on the ROC curve with gradient β defined as:

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

Thresholding is somehow limited, in facts we can segment images much better by eye than through thresholding processes. We might improve results by considering image context. In order to do this we first need to define what does it mean that two pixels are connected, we have two main options: either we consider 4-neighbourhood connectivity of 8-neighbourhood connectivity. Then we can do region growing, i.e. we start from a seed point and we add neighbouring pixels that satisfy the criteria defining the region. We repeat this process until we can include no more pixels. We have to take two major decisions:

- **Seed selection:** point and click seed point; select a seed region (either by hand or automatically from a conservative thresholding); multiple seeds.
- **Selection criteria:** greylevel thresholding; color or texture information; use mean and standard deviation of the region.

Another possibility is to use foreground-background segmentation, i.e. we do something like:

$$|I - I_{bg}| > T$$

where I_{bg} is the background image and T a threshold. We note that taking a background image is not easy (because even the background might not be static because of lighting changes, wind, clouds, ...) and the image does not only change because of the objects but there might also be secondary changes (e.g. the shadow of the object we want to segment). A better variant would be to fit a Gaussian model per pixel and taking into account statistical properties of the image:

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

where Σ is the background pixel appearance covariance matrix.
Another method to do this kind of classification is the use of Markov Random Fields. In this case we represent the picture as a graph where every pixel is a node and we have edges based on the pixel connectivity (e.g. if we use 4 pixel connectivity our graph is a grid). In this graph every edge has a label (based on the class of the pixel) we define a cost function as follows:

- We define the cost of an edge from v_i to v_j as 0 if the nodes are in the same class (basically, we want the classification to be as smooth as possible) and a constant K if the class is different.
- We define the cost of a node to be inversely proportional to the probability that our classification is correct.

The goal is to minimize the cost function (because this would mean that we have classified everything correctly with high probability and that the classification is smooth. This problem can be solved with a min cut approach.

Morphological operators

Local pixel transformations for processing region shapes used mostly on binary images. Logical transformations based on comparison of pixel neighbourhoods with a pattern. The goals are: smooth region boundaries for shape analysis; remove noise and artefacts from an imperfect segmentation; match particular pixel configurations in an image for simple object recognition.
Morphological operators take two arguments:

- A binary image
- A structuring element

In order to understand what they do we need some basic definitions:

- S **fits** I at x if:

$$\{y | y = x + s, s \in S\} \subset I$$

- S **hits** I at x if:

$$\{y | y = x - s, s \in S\} \cap I \neq \emptyset$$

- S **misses** I at x if:

$$\{y | y = x - s, s \in S\} \cap I = \emptyset$$

Erosion:
If S is the structuring element for eight connected neighbour: *Erase any foreground pixel that has one eight connected neighbour that is background.*
The image $E = I \ominus S$ is defined as:

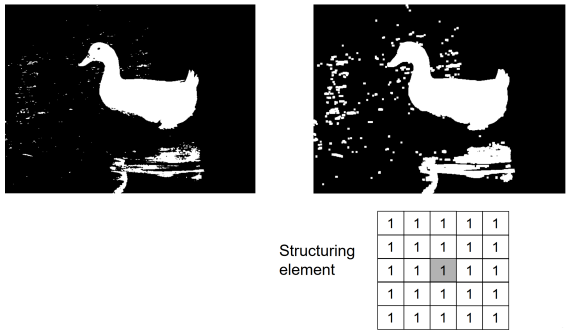
$$E(x) = \begin{cases} 1 & \text{if } S \text{ fits } I \text{ at } x \\ 0 & \text{otherwise} \end{cases}$$

Example

Dilation:
If S is the structuring element for eight connected neighbour: *Paint any background pixel that has one eight-connected neighbour that is foreground.*
The image $D = I \oplus S$ is defined as:

$$D(x) = \begin{cases} 1 & \text{if } S \text{ hits } I \text{ at } x \\ 0 & \text{otherwise} \end{cases}$$

Example



Opening:
The opening of I by S is:

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

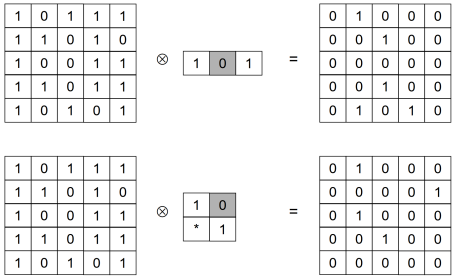
Small part of foreground become background.
Closing:
The closing of I by S is:

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

Small part of the background become foreground.

Hit-and-miss transform:
$H = I \otimes S$. Searches for an exact match of the structuring element, simple form of template matching.

Hit-and-miss transform



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

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

Convolution and filtering

Image filtering: modify the pixels in an image based on some function of a local neighbourhood of the pixels. We begin with **linear shift-invariant filtering**, which is about modifying pixels based on neighbourhood (i.e. to modify a given pixel we consider only pixel which are not too far away from the original pixel). Linear means that we do a linear combinations of neighbours. Shift-invariant means we do the same operation for every pixel. In general linear operations can be written as:

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

where:

- I input image
- I' output image
- K kernel of the operation
- N(m, n) a neighbourhood of (m, n)

It holds that operations are shift-invariant if K does not depend on (x, y).

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)

Correlation: we apply a kernel to the picture in a "natural way", i.e.:

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

for example:

$$\begin{aligned} &K(-1,-1)I(x-1,y-1) + K(-1,0)I(x-1,y) \\ &+ K(-1,1)I(x-1,y+1) + K(0,-1)I(x,y-1) \\ &+ K(0,0)I(x,y) + K(0,1)I(x,y+1) \\ &+ K(1,-1)I(x+1,y-1) + K(1,0)I(x+1,y) \\ &+ K(1,1)I(x+1,y+1) \end{aligned}$$

Convolution: we apply a kernel in a "mirrored way", i.e.:

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

for example:

$$\begin{aligned} &K(1,1)I(x-1,y-1) + K(0,1)I(x,y-1) \\ &+ K(-1,1)I(x+1,y-1) + K(1,0)I(x-1,y) \\ &+ K(0,0)I(x,y) + K(-1,0)I(x+1,y) \\ &+ K(1,-1)I(x-1,y+1) + K(0,-1)I(x,y+1) \\ &+ K(-1,-1)I(x+1,y+1) \end{aligned}$$

Convolution is a correlation with the kernel reversed with respect to the origin. If $K(i,j) = K(-i,-j)$ then correlation and convolution are equivalent.

We say that a kernel is separable if it can be written as:

$$K(m,n) = f(m)g(n)$$

This brings a computational advantage because, instead of applying a two dimensional filter, we can apply two different one dimensional filters sequentially. Rank 1 matrices represent separable filters.
The Gaussian filter is a filter which can be filled with the following formula:

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

and it takes into account the value of a pixel based on the distance from the other pixels (i.e. pixels which are closer to the original one have a greater influence).

The Gaussian filter is separable, in facts we have:

$$g(x,y) = \frac{1}{2\pi\sigma^2} exp[\frac{-(x^2+y^2}{2\sigma^2}] =$$
$$\frac{1}{2\pi\sigma^2} exp[\frac{-x^2}{2\sigma^2}] \frac{1}{2\pi\sigma^2} exp[\frac{-y^2}{2\sigma^2}] = g(x)g(y)$$

By the central limit theorem, if we convolve several times with a certain distribution we tend to a Gaussian. Hence, if we convolve a Gaussian with standard deviation σ with itself we get another Gaussian with standard deviation $\sigma\sqrt{2}$. Repeated convolution by a Gaussian filter produces the scale space of an image (i.e. you blur the image more and more according to how many times you convolve the image).

Other relevant filters are:

- **Prewitt operator:**

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

  useful to detect vertical edges.

- **Sobel operator:**

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

  also useful to detect vertical edges (but it takes more into account the central pixels and less the neighbours).

- **Laplacian operator:**

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

  The Laplacian filter is isotropic (rotation invariant). This filter wants to find the maximum of the first derivative or the zero crossing of the second derivative.

- **High-pass filter:**

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

High-pass filters let pass frequencies higher than a certain treshold, while low-pass filters let pass frequencies lower than a certain treshold. In the case of images, high pass filters makes images look sharper and they emphasize fine details. Since the high pass filter is sensitive to noise, one can first blur the image and then apply the filter.

Sharpening images increase the frequency components to enhance edge and they do the following operation:

$$I' = I + \alpha|k \cdot I|$$

where k is a high-pass filter and α is a scalar in $[0, 1]$.

Image features

Image features takes care of important parts of images (e.g. eyes in a face, ETH logo, edges, ...). The first thing that we discuss is **template matching**, i.e. recognizing an object described by a template $t(x, y)$ in the image $s(x, y)$. We search for the best match by minimizing the mean-squared error between image and template:

$$\begin{aligned} E(p,q) &= \sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty}(s(x,y)-t(x-p,y-q))^2 \\ &= \sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty}|s(x,y)|^2 + \sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty}|t(x,y)|^2 \\ &\quad - 2\sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty}s(x,y)t(x-p,y-q) \end{aligned}$$

which is equivalent to maximize

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

Hence in order to find the template in the image we first do the convolution of $s(x, y)$ with the impulse $t(-x, -y)$ and then we search the peak.

The next topic we discuss is **edge detection**. We know that an edge in an image is a region of the image where we have a big difference in the intensity. We do this by using some filters which sums to zero (intuitively because if we apply it to a region with uniform intensity we want to get zero as output). The idea (in a continuous space) is that there are edges where the value of the gradient operator is large (the gradient points in the direction of the maximum change and it's norm represent the slope of intensity change). We approximate the gradient with some discrete filters:

- **Prewitt:**

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

- **Sobel:**

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

- **Roberts:**

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Instead of using the gradient we could also use the Laplacian (i.e. sum of second order partial derivatives). This can be approximated with the following discrete filters:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

The Laplacian aims to detect zeros of the second order derivative (which are maxima of the first order derivatives) without considering zeros of the original intensity function. This type of filter is very sensitive to fine detail and noise, hence it is useful to blur images first, for example with a Gaussian filter. This brings us to the concept of the Laplacian of Gaussian (LoG) operator, which has the form:

$$LoG(x,y) = \frac{1}{\pi\sigma^4}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right)e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Another very important edge detector is the Canny edge detector, which is composed by 5 steps:

- Smooth image with Gaussian Filter
- Compute the magnitude and angle of the gradient with some filter such as Sobel, Prewitt, ... The formula for magnitude and gradient are given by:

$$M(x,y) = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$
$$\alpha(x,y) = \tan^{-1}\left(\frac{\partial f}{\partial x}/\frac{\partial f}{\partial y}\right)$$

- Apply nonmaxima suppression to gradient magnitude image. This means that if our filters detects more spots which look like an edge we keep only the ones with a very big response (only the maxima spots which look like an edge). This is useful because we know the direction of the edge (and we don't want to suppress things in this direction). Concretely we quantize edge normal to one of four directions (horizontal, vertical or in betweeen) and if $M(x, y)$ is smaller than either of its neighbours in edge normal direction we suppress it, otherwise we keep it.
- Double tresholding to detect strong and weak edge pixels.
- Reject weak edge pixels not connected with strong edge pixels.

The Hough transform can be used to extract different kind of features (e.g. lines, but also circles and other shapes). Let's begin with explaining the idea with lines. If we know the position (x, y) of an edge pixel we can represent all possible lines $y = mx + c$ which pass from (x, y) in a plane with parameters m and c as axes. If several edge pixels lie on a line in the original image, we will have a peak in the plane with parameters m and

c. Detecting the peak allows us to reconstruct the line which connects those pixel edges. This approach has some problem because we can not represent vertical lines (those would have the parameter m which tends to infinity) and hence, instead of having the parameters m and c, we could have polar coordinates. Hough transform can also be used to detect other shapes, e.g. circles. We could have a plane with parameters x' and y' which represent all possible circles with a given fixed radius that pass from a given point or a three dimensional hyperplane where we include a variable radius as third variable.

Apart from edges, it is also important to **detect corners**. Hence it is desirable to have a corner detector with accurate localization, invariance against shift, rotation, scale, brightness and robust against noise. To recognize edges the local displacement sensitivity can be useful:

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

where, for small Δx, Δy, we can use the following linear approximation:

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

where $f_x(x,y)$ is the horizontal image gradient and $f_y(x,y)$ is the vertical image gradient. We can then approximate the local displacement sensitivity by:

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

where

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

In order to find corners we want to maximize the eigenvalues of M: in facts if both eigenvalues are small it means that there is not a big difference of intensity; if only one of the two eigenvalues is large then we are in an edge (because the intensity changes significantly only in one direction) and if both eigenvalues are large then the intensity changes significantly in both directions and hence we are in a corner. In order to detect corners we can use Harris Formula:

$$\begin{aligned} C(x,y) &= det(M) - k(\text{ trace } (M))^2 \\ &= \lambda_1\lambda_2 - k(\lambda_1+\lambda_2)^2 \end{aligned}$$

where k is a small constant (e.g. 0.4).
Some desirable properties of corner detectors are:

- Accurate localization
- Invariance to shift, rotation, scale and brightness change
- Robust against noise
- High repeatability

The Harris corner detector is invariant to brightness change, to shift (because we use fixed size windows), to rotation (because eigenvalues don't depend on the rotation of ellipses and iso-sensitivity curves are ellipses), but is not invariant to scaling. This happens because it can happen that the corner does not fit into our fixed size window. A solution is to look for strong DoG responses over scale space.

Fourier Transform

There are some open questions which bring us to the concept of **Fourier transform**:

- What causes the tendency of differentiation to emphasize noise? Why when we do edge detection we don't just detect edges but we also emphasize noise?
- In what precise respects are discrete images different from continuous images?
- How do we avoid aliasing? Aliasing is the phenomenon that causes that if we want to shrink an image and in order to do that we just take every second pixel we get strange phenomena. Another way to think about aliasing is the clock: if we look at the seconds bar and we take a snapshot every 55 seconds, when we watch the sequence we have the impression that the bar goes backwards.

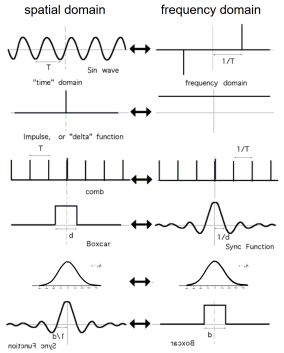
The Fourier transform represent the function in a new basis. In facts we can think to $n \times n$ images as a vector space of size n^2. We can apply a linear transformation to transform the basis into another one by doing a dot product between a transformation matrix and every basis element. The Fourier transform is a transformation where every component, instead of representing the intensity in a given pixel of the image, represents a wave pattern over the image. We will do this in a way that does not lose any information, in facts our matrix to change the base in full rank and hence invertible. This allows us to go from the pixel space to the Fourier space in both directions. The basis elements of the Fourier basis are of the form:

$$e^{-i2\pi(ux+vy)} = \cos 2\pi(ux+vy) - i\sin 2\pi(ux+vy)$$

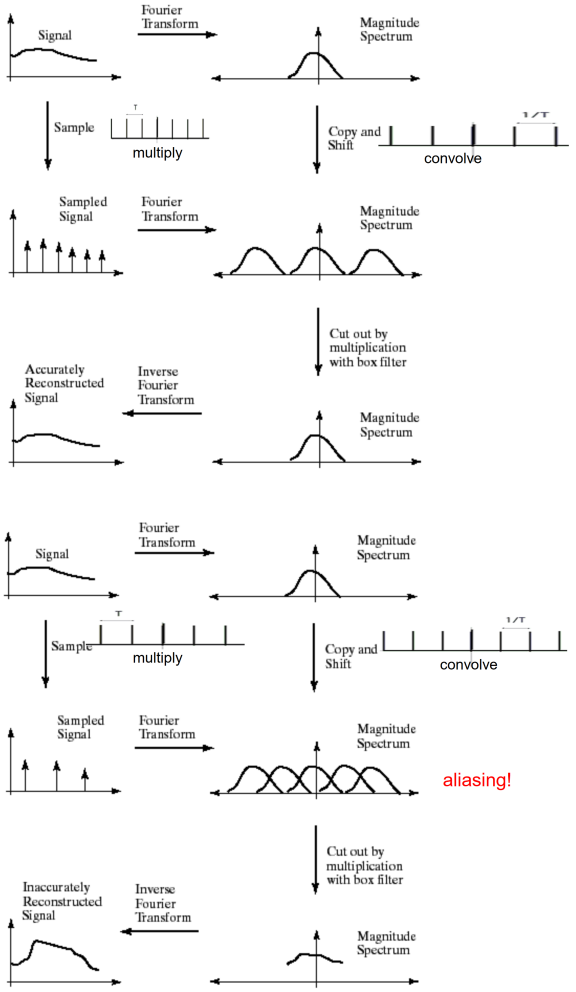
- $F(u) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi ux}dx$
- $f(x) = \int_{\infty}^{\infty} F(u)e^{i2\pi ux}du$
- $F(u,v) = \int_{\infty}^{\infty}\int_{\infty}^{\infty} f(x,y)e^{-i2\pi(ux+vy)}dudv$
- $f(x,y) = \int_{\infty}^{\infty}\int_{\infty}^{\infty} F(u,v)e^{i2\pi(ux+vy)}dudv$
- $F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} g(x,y)e^{-2\pi i\left(\frac{ux+vy}{N}\right)}$
- Convolution: $(f*g)(t) = \int_{\infty}^{\infty} f(\tau)g(t-\tau)d\tau$
- Convolution theorem: $F(g\cdot h) = F(g) * F(h)$ and $F(g*h) = F(g)\cdot F(h)$

The Fourier transform of a real function often has complex coefficients and for this reason is not easy to plot and visualize it. Instead we can represent it with magnitude and phase. A curious fact is that images have similar magnitude and very different phases. Let's see some important facts about the Fourier transform:

- Magnitude: $|F(u,v)|$
- Phase: $\tan^{-1}\left(\frac{Im(F(u,v))}{Re(F(u,v))}\right)$
- The Fourier transform is linear
- $F(f(bt))(\lambda) = \frac{1}{b}F(f)(\frac{\lambda}{b})$
- $F(f(t-a))(\lambda) = e^{i2\pi a\lambda)}F(f(t))(\lambda)$
- $F(f^n)(\lambda) = (i2\pi\lambda)^n F(f)(\lambda)$
- $F(f(ax+by)) = \frac{1}{abF(\frac{u}{a},\frac{v}{b}}$
- $F(f(x-a,y-b)) = e^{i2\pi(au+bv)}F(u,v)$
- $F(\delta(x-x_0))(u) = e^{-i2\pi u x_0}$
- $\delta(u) = \int_{-infty}^{\infty} e^{-i2\pi ux}dx$
- There is an inverse Fourier transform
- Scale function down means scale the transform up



Relation to sampling:



This means that in order to avoid aliasing we have to sample at least at the double of the maximum frequency (as formalised by Nyquist theorem).

Unitary transformations

A digital image can be represented with a matrix:

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

where we denote $f(x,y) = f_{yx}$. The matrix f can be represented as a column vector in the following way:

$$\vec{f} = \begin{pmatrix} f(0,0) \\ f(1,0) \\ \vdots \\ f(N-1,0) \\ f(0,1) \\ f(1,1) \\ \vdots \\ f(N-1,1) \\ \vdots \\ f(0,L-1) \\ \vdots \\ f(N-1,L-1) \end{pmatrix} = \begin{pmatrix} f_{00} \\ f_{01} \\ \vdots \\ f_{0(N-1)} \\ f_{10} \\ f_{11} \\ \vdots \\ f_{1(N-1)} \\ \vdots \\ f_{(L-1)0} \\ \vdots \\ f_{(L-1)(N-1)} \end{pmatrix}$$

Any **linear image processing** algorithm can be written as:

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

Where H is a matrix (not necessarily square, e.g. if we go from an image $N \times N$ to an image $M \times M$. We recall the definition of a linear operator L:

$$L\left[\alpha\vec{f_1} + \beta\vec{f_2}\right] = \alpha \cdot L\left[\vec{f_1}\right] + \beta \cdot L\left[\vec{f_2}\right]$$

Some linear algebra notations:

- $A^* = A$ if A is a real matrix.
- $A^* = \bar{A}$ if A is a complex matrix.
- A real matrix is symmetric iff $A^T = A$.
- A real matrix is ortogonal iff $A^{-1} = A^T$.
- A complex matrix is hermitian iff $(A^*)^T = A$.
- A complex matrix is unitary iff $(A^*)^T A = A\,(A^*)^T = I_n$.
- We can write $(A^*)^T$ as A^H.

Unitary transforms are transforms described by a unitary matrix, i.e. a matrix such that $A^{-1} = A^H$ (or A^T if the matrix is real). Those kind of matrices we have a **conservation of the energy or of the norm** and hence we can interpret them as a rotation of the coordinate system (and, possibly, sign flips). This holds because if $c = Af$ we have:

$$||c||^2 = c^H c = (Af)^H Af = f^H A^H Af = f^H f = ||f||^2$$

Eigenfaces: we want to solve the problem of face recognition, i.e. we have a database with several faces and, given a new face, we want to find which person it corresponds to (of course, provided that the face is into the database). A possibility to do this would be to consider the whole new image and calculate the euclidean distance between this image and all other images is the database. This is possible but very inefficient since images have a lot of pixels. In order to be efficient we use this method:

1. We represent all images in the database as a column vector and we represent them in a matrix A.
2. Those images must be normalized (i.e. all faces must be centered).
3. We remove from every column the average face in the database.
4. We calculate AA^T. This is the covariance matrix of A. The eigenvectors of the covariance matrix corresponding to the largest eigenvalues represent the direction along which the dataset has the maximum variance. Therefore they encode the features which differ the most among faces. We took k of those eigenvectors, which are also called eigenfaces (because if we represent this vectors as images they encode some face features). Every image can be expressed as the sum of the mean face plus a linear combination of those eigenfaces. In order to calculate the eigenvectors we can do the SVD decomposition of A. The eigenvectors of AA^T are the columns of matrix U.
5. For every image in the data set we calculate a vector p with the coefficients of the linear combination with the eigenfaces to represent the picture.
6. Given a new image we calculate the coefficients and we do an euclidean nearest neighbour with the images in the data set.
7. If the euclidean distance with the nearest neighbour is below a certain treshold we say that we have a match, otherwise not.

An advantage of this method is that we discard some noise in the pictures and in general works well because only main characteristics are preserved and irrelevant details are discarded. A limitation of this method is that differences due to varying illumination can vary across pictures and can become a bottleneck.

Linear discriminant analysis: imagine that we have a drug for a given illness. This drug works great for some people, but make other people feel worse. We want to decide who to give the drug to. In order to make a classification of the people we look at some gene expressions. We could take a look at a separation by using one, two, three or tausends of genes. However this might be inefficient. Linear discriminant analysis focuses on reducing dimensions by maximizing the separability among categories. Imagine that we want to project a two dimensional graph in a one dimensional graph in a way that maximizes the separability: a naive way could be ignoring one of the two components, but this is not very good because we discard all useful informations brought by that component. We want to find the axis which maximizes the separability and project all the points into this axis. This new axis is created by considering two criteria simultaneously:

- Maximize the distance between means
- Minimize the variation (scatter) within each category

In order to consider them together we say that we want to maximize the following ratio:

$$\frac{(\mu_1 - \mu_2)^2}{s_1^2 + s_2^2}$$

Fisherfaces are a method which takes into account both the distance between means (between-class scatter) and the within-class scatter. Formally we want to find the basis vectors (fisherfaces) which maximize:

$$W_{opt} = \arg\max_W \left(\frac{det\left(WR_BW^H\right)}{det\left(WR_WW^H\right)} \right)$$

with:

$$R_B = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^H$$

$$R_W = \sum_{i=1}^{c} \sum_{x_k \in Class(i)} (x_k - \mu_i)(x_k - \mu_i)^H$$

The matrix W which maximizes the equation is given by:

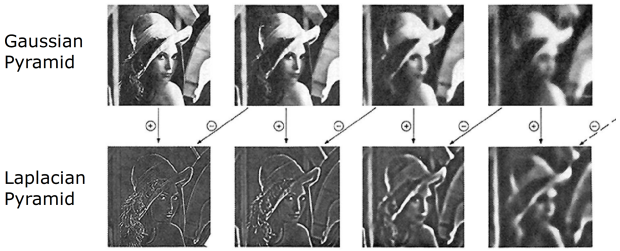
$$R_B w_i = \lambda_i R_W w_i$$

By using this method we don't preserve the maximum energy, but we project the faces in a space where they are optimally distinguishable. In general fisherfaces have a much higher recognition rate than eigenfaces.

Pyramids and Wavelets

The idea at the basis of pyramids is to exploit scale-space representations. That is from an original signal $f(x)$ (our original image), we generate a parametric family of signals $f^t(x)$, where fine-scale information is successively suppressed. That is from our original image we generate another image which contains less details and then we go on until a certain number of times. Image pyramids is the idea by which we express that from an original image we use some approximation filter which removes some fine-details and then we make the image smaller (usually by a factor of 2). Then from this image we go on recursively until we get to an image of a single pixel which contain some very coarse grained information about the original image. This approach can be used for correspondence searching (i.e. we first look at coarse scales and then we refine with finer scales by exploiting the information we gained at a coarser level) and edge tracking (i.e. a good edge at a fine scale has parents at a coarser scale). It was used for example in CMU edge detection. Two examples of pyramids are the following:

- **Gaussian pyramid:** smooth with Gaussian filter and then reduce the size. This representation is redundant because Gaussians are low pass filters and so we don't at a coarser level we don't loose many information.
- **Laplacian filter:** with this term we mean the difference of Gaussian. This is a band pass filter, i.e. each level represents spatial frequencies (largely) unpresented at other levels.



JPEG compression:

1. Transform RGB to YCbCr (because we are more sensitive to brightness change than color change).
2. Subsampling+partitioning: decrease chroma pixels (e.g. a 4×4 grid of the color image becomes one pixel). We divide the image in blocks of size 8×8.
3. DCT to frequency domain. DCT is similar to Fourier, but it does not involve complex numbers. We do DCT on the 8×8 blocks. The idea is that big objects have small frequency and small objects have big frequency. The top left pixel of the 8×8 block is called DC, the other ones AC.
4. Quantization: divide 8×8 blocks with elements of a quantization matrix. The elements of the quantization matrix are close to 1 in the top left area and big by going towards bottom right. We discard the zeros after the division (corresponding to high frequencies).
5. DC coding
6. RLE for AC
7. Huffman coding

Optical Flow

Optical flow studies how things moves in images and videos. Motion is important because sometimes is the only cue we can have for segmentation (e.g. random images with some patterns that move). An intuitive definition of optical flow is the apparent motion of brightness patterns. Ideally the optical flow is the projection of three-dimensional velocity vectors on image. However, this is not always true, for example we can have:

- Uniform sphere rotating. We will get an optical flow of zero although there is motion.
- Static scene with light changing. We will get a non-zero optical flow although there is no motion.

Applications:

- Motion segmentation: the image is segmented by looking at which parts of the image move differently.
- Stabilization: movement of the camera can be filtered out by using optimal flow in the captured video to calculate the movement of the camera and canceling it out.
- Tracking (follow something on a video sequence): an object in a video can be tracked out by using optical flow to follow it.
- Slow motion.
- Video compression: for video compression we can make use of the temporal redundancy and predict frames based on previously encoded frames using its optical flow. Video compression using optical flow is ineffective if there are many scene changes or high motion.

With $I(x, y, t)$ we mean the brightness at the point (x, y) at time t. The brightness constancy assumption tells that a pixel tat moves from one step to the other does not change its brightness, in a formula:

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

from which one can derive the optical flow constraint:

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

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.

Lukas-Kanade
Assumptions:

- Brightness constancy: projection of the same point looks the same in all frames.
- Small motion: objects move very slowly from frame to frame, which means that corresponding points from two consecutive images are not far apart
- Spatial coherence: points move like their neighbours. More precisely: points inside a patch move in the same way (reasonable, point which are part of the same object move in the same way).

The optical flow is computed for an image patch instead that for a single pixel. This is done in order to resolve the ambiguity of the optical flow equation $I_x \cdot u + I_y \cdot v + I_t = 0$, which is an equation with two unknowns.
The derivation works as follow: the brightness constancy assumption allow us to write

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t)$$

The small motion assumption allow us to use a Taylor approximation and we get:

$$I(x, y, t) \sim I(x, y, y) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t$$

Hence we want:

$$\frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t \sim 0$$

With the spatial coherence assumption we exploit the fact that multiple points move similarly and we get:

$$\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ \vdots & \vdots \\ I_x(p_n) & I_y(p_n) \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} -I_t(p_1) \\ \vdots \\ -I_t(p_n) \end{bmatrix}$$

If the equation on the left is invertible there is no problem, but if all gradients point in the same direction (e.g. when we have only one point or when we are along an edge) the matrix is not invertible and we have an aperture problem.
This approach works well for small motion. If we have large motion we can use coarse-to-fine estimation: we build image pyramids (the layers are the same image with different resolutions) and estimate the flow on each pyramid layer. Result of the flow calculation on the lowest resolution layer are passed to a higher resolution layer and so on. This works because large motion in the original image corresponds to smaller motion in the layers which have lower resolution. Thus, optical flow can be applied on those levels.

Iterative refinement

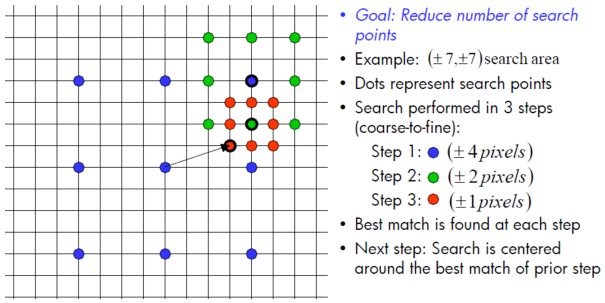
- Estimate velocity at each pixel using one iteration of Lucas and Kanade estimation.
- Warp one image towards the other using the estimated flow field.
- Refine estimate by repeating the process.

Video Compression

Our eye system is designed to look in a specific place. If we know where people would look in an image we could represent that area with high resolution and the rest of the picture with a lower resolution, if people look only in the area with high resolution this does not make a difference. However, since we don't know where people will look we can not do this. Video compression has some difference compared to image compression. In facts our visual system automatically follows motion (i.e. if we watch a video we will automatically focus on the sections of the sequence of images which move). This implies that some distorsions are not as perceivable as in image coding (but they become perceivable if we froze the video to a specific image). A succession of images is perceived as continuous if frequency is sufficiently high (e.g. 24Hz, but we would need some more in order to avoid aliasing and flickering, which can be perceived up to 60Hz in periphery).
A video is a 2D+t sequence (that is a sequence of images which change over time). A common way to compress video is to interlace it. Each frame of an interlaced video signal shows every other horizontal line of the image. As the frames are projected on the screen, the video signal alternates between showing even and odd lines. When this is done fast enough, i.e. around 60 frames per second, the video image looks smooth to the human eye. Because only half the image is sent with each frame, interlaced video uses roughly half the bandwidth than it would sending the entire picture. Videos can be compressed by dropping unimportant details, i.e. by exploiting spatial correlation between neighbouring pixels and temporal correlation between frames (in facts, most frames don't change a lot in the next time step). Hence there are two kind of redundancies that we can exploit in order to compress videos: temporal redundancy (which happens when two successive frames are very similar and there is just a slight motion between them) and non-temporal redundancy (i.e.techniques used when temporal redundancy is difficult to exploit, e.g. where there is a scene change or when there is high motion such as in an ice hockey game).
Predictive methods predict the current frame based on previously coded frames and there are three types of this:

- I-frame: frame coded independently of other frames (just like an image).
- P-frame: code based on previously coded frame.
- B-frame: coded based on both previous and future coded frames

A popular technique is called **block matching motion estimation**. The idea is that we partition each frame into blocks of a given size and then we describe the motion of each block by finding the best matching block in the referenced frame. For each block we have a motion vector which describes the translation of the box from the old frame to the new one. The collection of motion vectors for all the blocks in a frame is the motion vector field. In order to decide which one is the best block in the new frame we have two different metrics: MSE (which minimizes the square of the difference between blocks) and MAE (which minimizes the absolute value). In order to search for the best block we could use a full search (which can be inefficient), or a partial search such as 3-step log search which is shown in the following image.



The advantages of this algorithm are: robustness to compression; motion vector field is easy to represent; simple and periodic structure. The disadvantages are: breaks down for complex motion (i.e. non translational); often produces blocking artifacts.
This technique, combined with things which are used in image compression such as DCT to exploit spatial redundancy, color space conversion, quantizer and huffman coding, allow one to build a good video compression method.

Texture

We represent texture as a weighted sum of some dictionary words and we want to enforce sparsity instead of smoothness. In texture synthesis we have an example of the texture that we want to generate and the goal is to generate another image which is similar but not exactly the same (i.e. you don't copy the pattern but you generate a similar statistic).
Texture can be done with pyramids by building a Laplacian pyramid of the texture image and then apply some

oriented filter to each level. This gives us image information at a particular scale and orientation. This is better then using different sized filters for efficiency (smaller filters on smaller images are more efficient than big filters on big images).
Histograms are not good for textures because they give only frequency information and no positional information. Moreover many different textures have the same histogram. A better solution is using the co-occurrence matrix.
In order to generate texture maps one can use the chaos-mosaic technique, i.e. use a file with the final image of the texture and choose random blocks of image and place them in random spots. This works well for textures with smooth edges (e.g. vegetation), but in some cases it may bring troubles (e.g. brick wall).

Radon Transform

In medical imaging we can distinguish two forms of imaging methods:

- Radiation source is outside the body (e.g. X-ray, ultrasounds)
- Radiation source is inside the body (e.g. MRI, PET, SPECT)

We will take a look at computed tomography (CT) which use a mathematical basis developped by Radon in 1917 but became a scanning device only in the 1960s. The idea is that x-rays pass through the body from different directions and, by knowing the difference of the energy when the ray enters the body and when the ray leaves the body, we have a measure of the tendency of the object to absorb or scatter x-rays. Since different materials absorb the rays in a different way, by combining multiple measurements we can get insights about the structure of the object. CT has had the following evolution:

- **Parallel beam:** a geometry with a single pencil which is shooting parallel X rays through the object which were detected on the other side. The system is moving in a circle.
- **Fan-bean:** X-rays are emitted under angle and collected by a 1D array of detectors.
- **Cone-beam:** use a cone shape and a 2D array detector.

An X-ray moves along a straight line and at a position s has intensity $I(s)$. When the X-ray travels a little bit through the object (by a δs) the intensity is reduced by δI. The reduction depends on the intensity and on the optical density $u(s)$ of the material. For small δs we have:

$$\frac{\delta I}{I(s)} = -u\delta s$$

By combining all the contributions to the reduction of the intensity of the X-ray we get:

$$I_f = I_0 e^{-R}$$

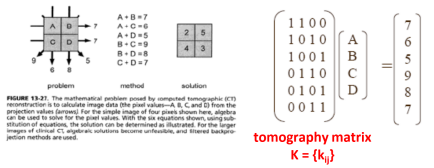
where R is the Radon transform of $u(s)$:

$$R = \int_L u(s)ds$$

The Radon transform has the following properties:

- Linearity.
- If the function is shifted, only the ρ coordinate changes, not the angle.
- If the function is rotated, only the Θ coordinate changes, not the radius.
- Convolution: The Radon transform of the 2D convolution of two functions is equal to the 1D convolution of the Radon transform of the same functions.

By measuring the intensities of the rays at the beginning and at the end we can know the value of the Radon transform. How can we go from the value of the Radon transform to the value of the function u?



Tomography system through the lense of linear algebra:

- Assumption - attenuation of material within each pixel constant and proportional to the area of the pixel illuminated by the beam

$$k_{ij} = \frac{\text{area of pixel } j \text{ illuminated by ray } i}{\text{total area of pixel } j},$$

$i = 1, \ldots, l,\ j = 1, \ldots, nm.$
Hence, the algebraic model reads

$$Kf = g,$$

f = BW plane/volumetric image to be retrieved (reshaped into a vector)
g = attenuation measurements from the CT system

The system can be solved in a least square sense.
An alternative approach is to exploit the Fourier Slice Theorem, i.e. the fact that the 1D Fourier transform of the attenuation measurements $g = Rf$ is equal to the 2D Fourier transform of the function. This gives us the following algorithm:

1. Measure projection (attenuation) data.
2. Do the 1D Fourier transform of projection data.
3. Make the 2D inverse Fourier transform and sum with previous image.

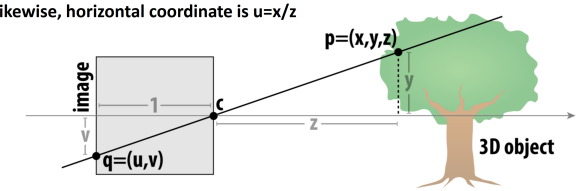
The previous algorithm gives a final image which is blurred, but this issue can be solved by applying an high-pass filter between points 2 and 3.

Introduction to graphics

Computer graphics is the use of computers to synthesize and manipulate visual information.
As introductory activity we considered drawing a 3D cube in the 2D plane. The cube is described by the coordinates of its vertices, by its edges and an observation point. The basic strategy is mapping 3D vertices into 2D points and then drawing straight lines between 2D points corresponding to edges. We have to consider the perspective, i.e. the fact that objects look smaller as they get further away.

Perspective projection: side view

- Where does a point p=(x,y,z) from the world end up in the image (q=(u,v))?
- Notice the two similar triangles!
- Assume camera has unit size, origin is at pinhole c
- Then v/1 = y/z (vertical coordinate v is just the slope y/z)
- Likewise, horizontal coordinate is u=x/z



With this in mind we can design a simple algorithm to draw the 3D cube in the plane. Let's suppose that the camera is in an arbitrary position $c \in \mathbb{R}^3$. We do the following:

1. For each edge (u, v) we compute $(u-c, v-c)$ which is of the form $((x_1, y_1, z_1), (x_2, y_2, z_2))$.
2. We get the following edges in the two dimensional space: $(\frac{1}{z_1}(x_1, y_1), \frac{1}{z_2}(x_2, y_2))$.

Now we consider another important problem: how to draw lines on a screen? Which pixels should we color in to depict the line? This is a first example of **rasterization**, i.e. conversion of continuous objects to a discrete representation on a pixel grid. Two examples to address this problem are:

- Color every pixel touched by the line.
- Diamond rule (i.e. color the pixel if and only if the line touches the *core* of the pixel).

Note that in principle it is possible to check for every pixel of the image, whether it should be colored or not, but this has complexity $\mathcal{O}(n^2)$ where n^2 is the number of pixels in the image. However there is a better method to address the problem, known as **incremental line rasterization**.
The idea is the following: a line can be represented by its endpoints (u_1, v_1) and (u_2, v_2). The line has slope $s = \frac{v_2 - v_1}{u_2 - u_1}$. Let's consider the special case where the line points up and right (the other cases are analogous). We can do the following:

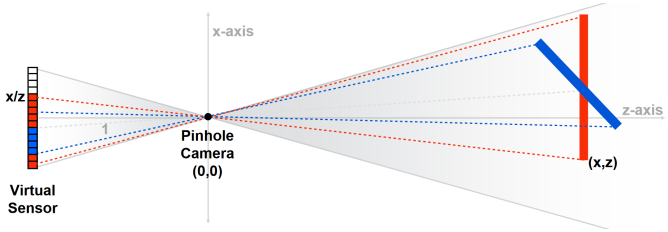
```
v = v1;
for( u=u1; u<=u2; u++ )
{
    v += s;
    draw( u, round(v) )
}
```

Drawing triangles

In order to draw triangles on the screen we have to answer two important questions:

- **Coverage:** given a pixel, is it in the triangle or not?
- **Occlusion:** if a pixel belongs to more than one triangle, which color should we use? In other words, which triangle is closest to the camera?

The visibility problem answers the question: *which scene geometry is visible within each screen pixel?*



We can approach this problem from two different perspectives:

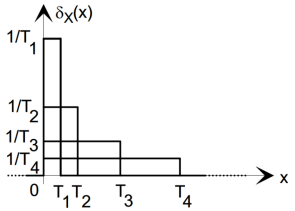
- What scene geometry projects into a screen pixel? (Coverage) Which geometry is visible from the camera at that pixel? (Occlusion)
- What scene geometry is hit by a ray from a pixel through the pinhole? (Coverage) What object is the first hit along the ray? (Occlusion)

Now we go back to the coverage problem, i.e. determine which pixels are overlapped by the triangle. There are several possibilities:

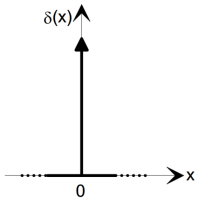
- The whole pixel is part of the triangle.
- The whole pixel is not part of the triangle.
- A part of the pixel is part the triangle, another part not.

If we are in one of the two cases the problem is easy, if we are in the third part we compute the fraction of the pixel which is covered by the triangle (with the same idea of Monte Carlo sampling) and we assign the corresponding fraction of the color of the triangle to the pixel. If we have more than one triangle in this pixel we do the sampling for every single triangle and we mix the results.
Now we consider again the problem of sampling (i.e. representing a continuous function in a discrete manner, which is exactly what we do with the representation of triangles on pixels). Obviously we can sample some points of a function f in a regular interval and then use some methods (e.g. nearest neighbour or linear interpolation). If we do it we see the tradeoff between number of samples and efficiency: if we increase the number of samples we get a better result, but the cost for it is a worse performance. The sampling rate is important, but it is limited in real world situations (e.g. we can not get the frequence of pixels that we want because this is limited by state-of-the-art technology). Now we go into some details of the mathematical representation of sampling. First we define the Dirac function:

$$\delta(x) := \begin{cases} 0 & \text{for } x \neq 0 \\ \text{undefined} & \text{at } x = 0 \end{cases}$$



Unit pulse functions



An Impulse

The Dirac function has the following property:

$$\int_{-\infty}^{\infty} \delta(x)dx = 1$$

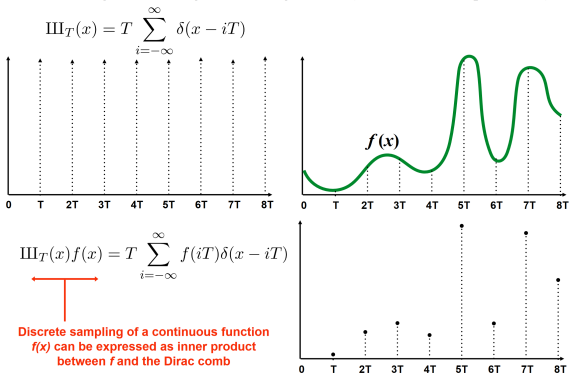
The sifting property of the impulse says that:

$$f(a) = \int_{-\infty}^{\infty} f(x)\delta(x-a)$$

Now we consider two problems: sampling (i.e. approximating a continuous function discretely) and reconstruction (i.e. given a discrete approximation of a function we have to reconstructing in the best possible way).

Sampling function

Consider a sequence of impulses with period T (Dirac comb or impulse train):



The reconstruction can be represented as a convolution between a filter and the sampled signal:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy$$

where some examples of filters are:

$$f(x) = \begin{cases} 1 & |x| \leq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$h(x) = \begin{cases} 1/T & |x| \leq \frac{T}{2} \\ 0 & \text{otherwise} \end{cases}$$

Now we can think to coverage as a 2D signal:

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

where we simply consider a coverage point in the pixel (e.g. pixel center). If we have edge cases (e.g. two triangles both have an edge on the coverage point), there are different possibilities. OpenGL/Direct3D say that an edge that falls directly on a screen sample point is classified within the triangle if the edge is a top or a left edge.

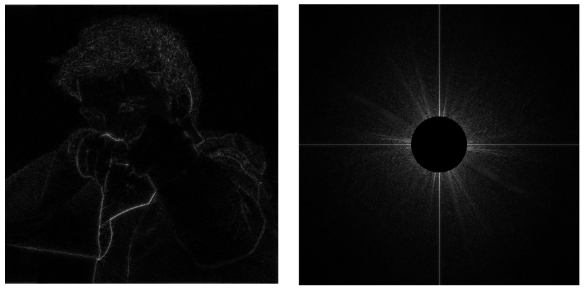
We know that signals can be represented as superpositions of frequencies (i.e. a function $f(x)$ can be represented as $f(x) = f_1(x) + f_2(x) + f_3(x)$). The same happens with images, let's see an example:

Low frequencies only

Spatial domain result

Spectrum (after low-pass filter)
All frequencies above cutoff have 0 magnitude

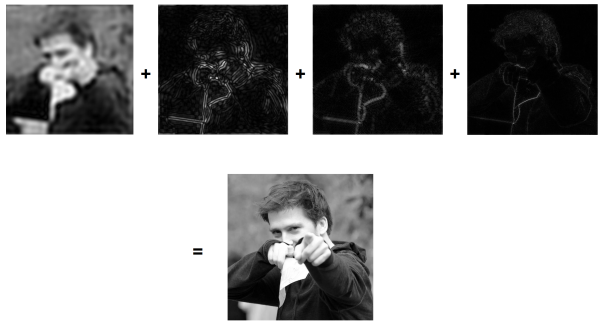
High frequencies (edges)



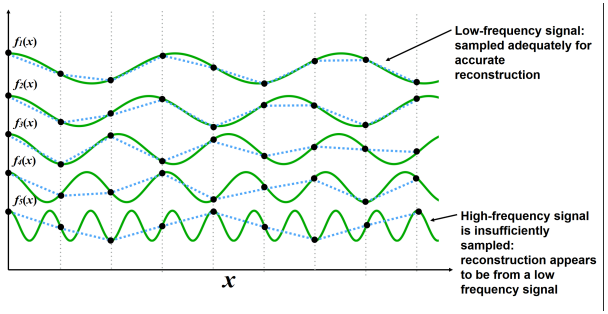
Spatial domain result
(strongest edges)

Spectrum (after high-pass filter)
All frequencies below threshold have 0 magnitude

An image as a sum of its frequency components



The problem of aliasing happens when we sample high frequencies but, due to undersampling, those high frequencies appears as low frequencies.



An intuitive example of aliasing is given by the clock. If we take a picture every 55 minutes, it looks like the clock hand goes backwards. In this context there is the **Nyquist-Shannon theorem**, which says that a signal where all frequencies are less equal than ω_0 can be perfectly reconstructed if sampled with period $T > \frac{1}{2\omega_0}$ and the reconstruction is performed using a normalized sinc. Aliasing happens a lot on images due to undersampling and hence we get strange artifacts in edges. A possible solution is oversampling.

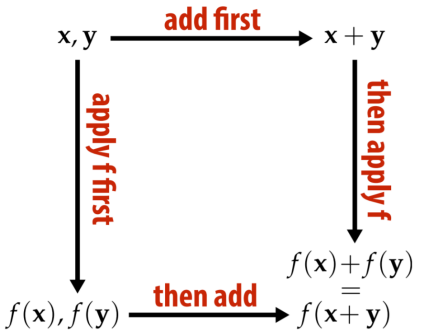
We conclude by explaining how to do a **point-in-triangle test**. Given the vertices $P_i = (X_i, Y_i)$ we compute $dX_i = X_{i+1} - X_i$ and $dY_i = Y_{i+1} - Y_i$. Then we compute $E_i(x,y) = (x - X_i)dY_i - (y - Y_i)dX_i$. If $E_i(x,y)$ is zero, then the point is on the edge, if it is greater than zero it is outside the edge and if it is less than zero it is

inside the edge. So in order to compute if a point is inside the triangle we check that it is inside all three edges (or on some edges based on the triangle coverage rules).

Transforms

In computer graphics transforms are everywhere. A transform is a function which maps a point of an image to another point. By applying this function to every point of a shape (e.g. a cube) we can do several things (make it taller, fatter, squishier, slantier, ...). In this course we study linear transforms because, computationally speaking, they offer several advantages and they are still very powerful. Moreover, over a short distance or small amount of time, all maps can be approximated as linear maps (Taylor's theorem). Composition of linear transformations is linear. The key idea of linear maps is that they maps lines to lines, while keeping the origin fixed. In order to check whether a map f is linear we have to show that for all vectors u, v and every scalar α we have:

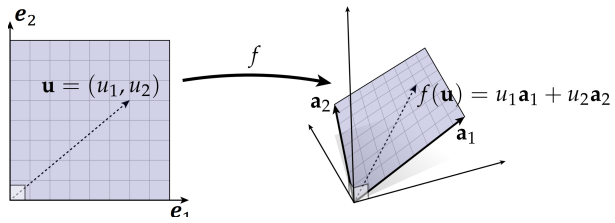
$$f(u+v) = f(u) + f(v)$$
$$f(\alpha u) = \alpha f(u)$$



For maps between $\mathbb{R}^m$ and $\mathbb{R}^n$ we have that if a map f can be expressed as

$$f(u) = \sum_{i=1}^{m} u_i a_i$$

with fixed vectors a_i, then it is linear. In this case all $a_i \in \mathbb{R}^n$. It is easy that if the formula above holds, then the requirements to show that a map is linear are satisfied.



We have:

- u is a linear combination of $e_1 = \begin{pmatrix} e_{11} \\ e_{12} \end{pmatrix}$ and $e_2 = \begin{pmatrix} e_{21} \\ e_{22} \end{pmatrix}$ i.e. $u = \alpha \cdot e_1 + \beta \cdot e_2$.
- Since f is a linear map we have: $f(u) = u_1 a_1 + u_2 a_2$. We can rewrite this as $f(u) = f(\alpha \cdot e_1 + \beta \cdot e_2) = f\left(\begin{pmatrix} \alpha \cdot e_{11} \\ \alpha \cdot e_{12} \end{pmatrix} + \begin{pmatrix} \beta \cdot e_{21} \\ \beta \cdot e_{22} \end{pmatrix}\right) = (\alpha \cdot e_{11} + \beta \cdot e_{21}) \cdot a_1 + (\alpha \cdot e_{12} + \beta \cdot e_{22}) \cdot a_2 = \alpha\,(a_1 \cdot e_{11} + a_2 \cdot e_{12}) + \beta\,(a_1 \cdot e_{21} + a_2 \cdot e_{22}) = \alpha \cdot f(e_1) + \beta \cdot f(e_2)$. This means that $a_1 = f(e_1)$ and $a_2 = f(e_2)$ and hence by knowing how to map the basis vectors of the initial basis we know how to map the entire space.

Now we recall a very useful formula from linear algebra. If we have a vector space whose basis can be described with a matrix S with respect to a reference space R and another vector space whose basis can be described with a matrix E with respect to R, then if we have a vector v expressed in the basis S we can transform it in basis E with the following formula:

$$E^{-1}Sv$$

More in general, if instead of using the identity map we want to use an arbitrary map M (always expressed with respect to R) we get:

$$E^{-1}MSv$$

For example consider the basis matrices $A = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & 3 \\ 3 & 1 \end{pmatrix}$ given with canonical coordinates. Consider the vector $v = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$ with respect to A. This means that the vector v is the vector $\begin{pmatrix} 6 \\ 4 \end{pmatrix}$ with respect to the canonical coordinates. In order to express v with respect to B we do:

$$\begin{aligned} & B^{-1}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} Av \\ &= \begin{pmatrix} -\frac{1}{8} & \frac{3}{8} \\ \frac{3}{8} & -\frac{1}{8} \end{pmatrix}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 2 \\ 2 \end{pmatrix} \\ &= \begin{pmatrix} \frac{3}{4} \\ \frac{7}{4} \end{pmatrix} \end{aligned}$$

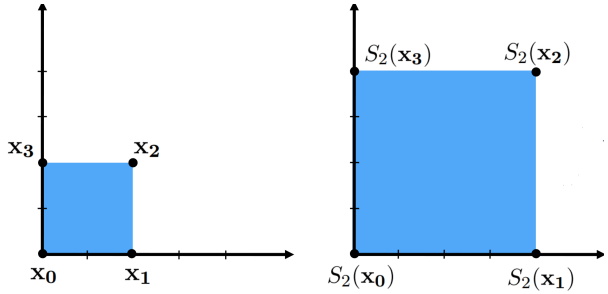
Now we are going to see several useful transformations:

Uniform scale:
We simply do the following operation for every vector x: $S_a(x) = ax$. In order to write this in matrix form we simply take a look to what happens to basis vectors and we get:

$$S_a(x) = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}$$

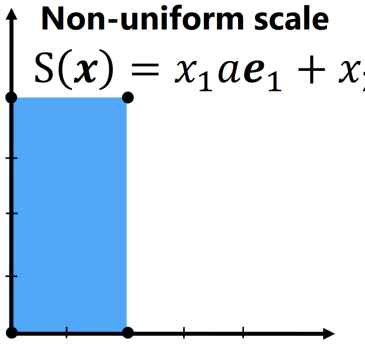
We show that is is a linear transformation:

$$S_a(\alpha x) = a\alpha x = \alpha(ax) = \alpha S_a(x)$$
$$S_a(x+y) = a(x+y) = ax + ay = S_a(x) + S_a(y)$$



Non uniform scale:
Very similar to uniform scale, but the constant by which we multiply the basis vectors is different, i.e. $S(x) = ax_1e_1 + bx_2e_2$ or, in matrix form:

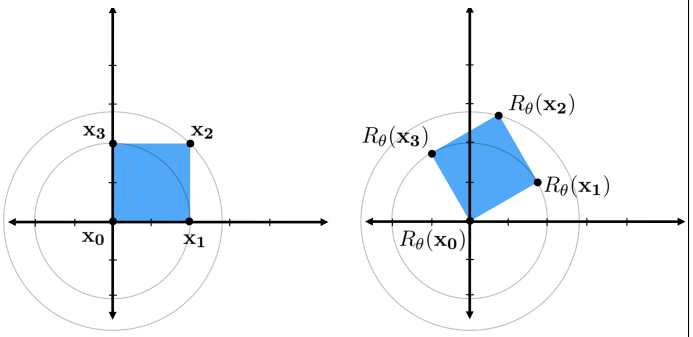
$$S(x) = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} x$$



Rotation:
Rotation means the rotation (in counter-clockwise direction) of every point by an angle Θ. This transformation preserves the norm of every vector (i.e. the distances between points in the shape don't change) and hence it is called isometric transformation. Again, in order to know how it looks like, we think to what happens to the basis vectors and we get $R_\Theta(x) = x_1 \begin{pmatrix}\cos(\Theta)\\ \sin(\Theta)\end{pmatrix} + x_2 \begin{pmatrix}-\sin(\Theta)\\ \cos(\Theta)\end{pmatrix}$ or, in matrix form:

$$\begin{pmatrix}\cos(\Theta) & -\sin(\Theta)\\ \sin(\Theta) & \cos(\Theta)\end{pmatrix}$$

Note that rotation is a linear transformation only if we do a rotation about the origin, otherwise (since linear transformations preserve the position of the origin), the transformation is not linear.



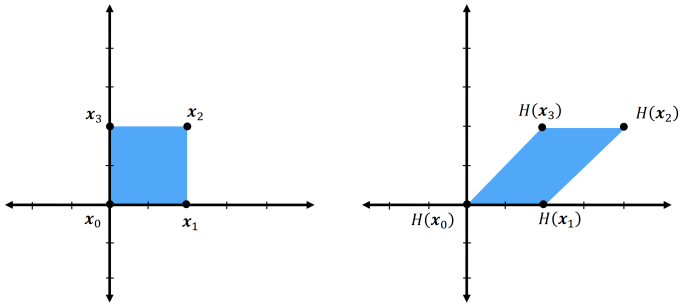
$$R_{x,\Theta} = \begin{pmatrix}1 & 0 & 0\\ 0 & \cos(\Theta) & -\sin(\Theta)\\ 0 & \sin(\Theta) & \cos(\Theta)\end{pmatrix}$$

$$R_{y,\Theta} = \begin{pmatrix}\cos(\Theta) & 0 & \sin(\Theta)\\ 0 & 1 & 0\\ -\sin(\Theta) & 0 & \cos(\Theta)\end{pmatrix}$$

$$R_{z,\Theta} = \begin{pmatrix}\cos(\Theta) & -\sin(\Theta) & 0\\ \sin(\Theta) & \cos(\Theta) & 0\\ 0 & 0 & 1\end{pmatrix}$$

Shear in x direction:
A transformation described by the following matrix:

$$\begin{pmatrix}1 & a\\ 0 & 1\end{pmatrix}$$



Now we take a look at the **translation** transformations that, for all vectors x returns the vector $x + b$ where b is a constant vector. This transformation is not linear, but it is affine. However there is a simple trick which makes this transformation linear, i.e. **homogeneous coordinates**. The key idea is to lift 2D points to a 3D space. Concretely we represent the point $\begin{pmatrix}x_1\\ x_2\end{pmatrix}$ as $\begin{pmatrix}x_1\\ x_2\\ 1\end{pmatrix}$. And 2D transforms are represented by 3×3 matrices, e.g. the rotation matrix is represented as:

$$\begin{pmatrix}\cos(\Theta) & -\sin(\Theta) & 0\\ \sin(\Theta) & \cos(\Theta) & 0\\ 0 & 0 & 1\end{pmatrix} \begin{pmatrix}x_1\\ x_2\\ 1\end{pmatrix}$$

Now we can represent translations in the following way:

$$\begin{pmatrix}1 & 0 & b_1\\ 0 & 1 & b_2\\ 0 & 0 & 1\end{pmatrix} \begin{pmatrix}x_1\\ x_2\\ 1\end{pmatrix} = \begin{pmatrix}x_1 + b_1\\ x_2 + b_2\\ 1\end{pmatrix}$$

and one can easily show that this is a linear map.
In general we can express several operations by composing different linear maps (e.g. to rotate with respect to a certain point different from the origin we first do a translation, then a rotation and finally another translation). It's important to note that the composition of maps is non commutative (because matrix multiplication is non commutative).

Geometry and Texture

What we have seen about transformations can be used in order to map an object in world coordinate to our final image. A very important concept is the one of the view frustum, which is basically the area that will be in the final image. The following matrix is the transformation to put everything in an unit cube:

$$\begin{bmatrix}\frac{f}{aspect} & 0 & 0 & 0\\ 0 & f & 0 & 0\\ 0 & 0 & \frac{zfar+znear}{znear-zfar} & \frac{2 \cdot xfar \cdot znear}{znear-zfar}\\ 0 & 0 & -1 & 0\end{bmatrix}$$

Hence in order to render an image captured with a camera in the 3D world we have the following pipeline:

1. Transform world coordinates to camera coordinates. Canonical form: camera at origin looking down the z-axis.
2. Projection transform + homogeneous divide. Canonical form: visible region of scene contained within the cube.
3. Screen transform.
4. Compute screen coverage.

Now we want to go beyond the cube man and do some more realistic geometry.
Geometry can be described in multiple ways:

- Linguistic, e.g. unit circle
- Implicit, e.g. $x^2 + y^2 = 1$
- Explicit, e.g. $(\cos(\Theta), \sin(\Theta))$
- Dynamic, i.e. with differential equations
- Discrete, i.e. by a set of points

Which option is the best depends on what we have to do, in facts there are some trade-offs. For example if we use implicit coordinates it is easy to know whether a point is part of the shape we are describing or not, but sampling (i.e. give me a point in the shape) is harder. On the other hand explicit coordinates are very useful to sample (because all points are directly given), but inside/ outside test can be difficult. Hence we can say that different representations are better suited for different types of geometry and different types of operations we may want to perform. Now we want to understand how to generate images which are not like the *cube man* but more realistic.
First, let's take a closer look to **implicit representations**:

- **Algebraic surfaces:** a surface is a zero set of a polynomial in x, y, z. However, to represent complicated shapes such as a cow, it is very difficult to come up with a polynomial. In order to do that we use constructive solid geometry, i.e. we build complicated shapes via boolean operations such as union, intersection and difference. By applying a chain of those operations we can build more complicated shapes.

- **Distance function:** gives distance to closest point on object.

- **Level set method:** implicit methods can be written in the form $f(x) = 0$ (i.e. every x that satisfies the relation is on the shape). We can store a grid of values of approximating the function f and then we say that the surface is found where interpolated values equal zero. Of course, if we use a grid with more points we have something more accurate. The drawback of this is that the storage for a 2D surface is now $\mathcal{O}(n^3)$ (also for an two dimensional image in the 3D space we need to approximate f in the whole space).

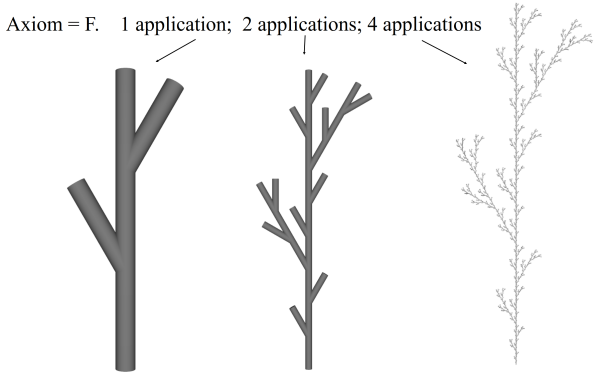
- **L-Systems:** very useful to represent fractals or organic shapes (e.g. trees). L-Systems are composed by a set of grammar rules, a start symbol and a semantic that maps every symbol to a concrete *action* in the image. For example one can have F as start symbol and a single grammar rule which maps F to $F+F--F+F$. In this case, after two applications, we get:

$$
\begin{aligned}
&F+F--F+F\\
&+\\
&F+F--F+F\\
&--\\
&F+F--F+F\\
&+\\
&F+F--F+F
\end{aligned}
$$

If we assign to F the meaning *draw forward one unit*, to $+$ the meaning *rotate left* and to $-$ the meaning *rotate right* we draw (when we apply the grammar rule several times), the von Koch snowflake curve.

We can add the branching structures in order to make the L-System even more powerful. This means that we use the square brackets to simulate a stack behaviour. Concretely we do the operations and when we see a [we save the state we are currently in and then we do the operations that follows the bracket until we see a]. When we get there we go back to the state we saved before entering in the first square bracket. By doing this we can easily model very organic shapes which look like a tree:

Example of this L system: F→F[+F]F[-F]F



Moreover, L-Systems can be made even better by adding randomness, i.e. for some starting symbols we have more than one grammar rule, each one with a given probability.

In general implicit methods have the following pros:

- Easy to make inside/ outside tests.
- Other queries may be easy, e.g. distance to surface.
- Easy to handle changes in topology.

And the following cons:

- Expensive to find all points in the shape.
- May be difficult to model complex shapes.

Now we move into some techniques for **explicit representations:**

- **Point cloud:** this is the simplest explicit representation and it consists of a set of points of the form (x, y, z). In this way we can easily represent any kind of geometry, but this is useful only for large datasets (i.e. when we have several points per pixel). With this method it's difficult to draw undersampled regions.

- **Polygonal mesh:** store vertices and polygons, mostly triangles. With this method it is easier to do processing and simulations, but they require more involved data structures. An example of data structure (used in the STL format), simply saves all triangles with a triple of vertices: this is redundant because we save some point more than once, but it is still used nowadays. An alternative is the indexed face set that assign to each vertex an ID and then it stores all triangles by using the IDs instead of writing the whole coordinate multiple times. There are multiple ways to generate a polygonal mesh, e.g. subdivision surfaces (i.e. applying an averaging rule in order to obtain a smooth surface from a starting polygon).

Until now we have seen how to represent a real world object into a screen. However we still can improve and the next topic we are going to discuss is light. In facts, when we look at a picture, not every area of an object has the same lighting conditions. In order to understand this we

do an analogy to what happens with seasons: why it's winter colder than summer? This has a relationship with the irradiance, i.e. the energy per time per area which is calculated as $E = \frac{\Phi}{A}$ where Φ is a measure for the energy that lands on A. Lambert's law tells that if we consider two different areas separated by an angle Θ we get:

$$E = \frac{\Phi}{A'} = \frac{\Phi \cos(\Theta)}{A}$$

Hence the reason why we have seasons has to do with the changing of this angle Θ during the year.
The simplest shading model in computer graphics is called N-dot-L lighting. This means that we multiply a gray scale image by the factor $\max(0, N\dot{L})$ where N is a vector normal to the surface and L is the direction of the light. Since the two vectors are normalized the dot product gives us the cosine of the angle between the normal to the surface and the light source. If the angle is in $\left[\frac{\pi}{2}, \frac{3\pi}{2}\right]$ the angle the surface is "the dark side of the moon" and hence is completely black, otherwise the value of the color is larger when the light comes in a more perpendicular way.
Texture: texture is used to map some images on a surface, e.g. to add a wood effect to the faces of a cube. Basically for each covered screen sample we must find the equivalent texture coordinate; then we lookup for the color in the texture image and we set the sample's color to the color we have found in the texture image. Usually we do this process for triangle vertices and then we use attribute interpolation for the samples inside the triangles. We start from samples which have the same distance in the image plane but variable distances in the texture space where we are actually doing the sampling. Hence we have to be very careful in order to not introduce aliasing artifacts.

Graphics Pipeline

Now we are going to put what we know (i.e. how to draw a triangle, transforms, perspective projection and texture sampling) together in order to get the end-to-end rasterization pipeline. The first topic we need to cover is **occlusion**.
If we have some image which is not yet rasterized (i.e. we haven't assigned colors to the pixels), how can we assign a color to each pixel based on which object is closer? A useful tool is the depth buffer (aka Z buffer), a two dimensional array that for each coverage sample point stores the depth of closest object at this sample point that has been processed by the render so far. By convention black means small distance and white means large distance. In order to compute the depth of sample points on a triangle we can save the depth values of the vertices and then interpolate just like any other attribute that varies linearly over the surface of the triangle. If we have three triangles and an occlusion phenomena the order by which we process the triangles has an impact on efficiency, but not on the correctness of the algorithm. The depth test works as follow:

1. Inputs: depth of sample point d; color of d; color matrix; coordinate x of sample; coordinate y of sample.
2. if depth of d is less than the depth stored in the Z buffer on the coordinates x,y, then update the coordinate of the z buffer on the coordinates x, y to the depth of d and assign to the color matrix in position x,y the color of d.

This approach also works with supersampling because the occlusion test is per sample and not per pixel.
Now we want to deal with semi-transparent objects and the tool we will use for this is **compositing**. We don't use only rgb value, but we add a parameter α for the opacity ($\alpha = 1$ means fully opaque, $\alpha = 0$ means fully transparent). With the help of this new parameter we can define the over operator (which is non commutative). B over A means: composite image B with opacity α_B over image A with opacity α_A. The composited color of the new image C is given by $\alpha_B B + (1 - \alpha_B)\alpha_A A$ and the opacity of the image C is $\alpha_B + (1 - \alpha_B)\alpha_A$. In order to be compact we can write:

$$A' = [\alpha_A A_R \quad \alpha_A A_G \quad \alpha_A A_B \quad \alpha_A]^T$$
$$B' = [\alpha_B B_R \quad \alpha_B B_G \quad \alpha_B B_B \quad \alpha_B]^T$$
$$C' = B' + (1 - \alpha_B)A'$$

If we want to render a mixture of opaque and transparent triangles we do the following:

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

This gives us the following pseudocode:

```
over(c1, c2) {
  return c1 + (1-c1.a) * c2;
}

update_color_buffer(tri_d, tri_color, x, y) {

  if (pass_depth_test(tri_d, zbuffer[x][y]) {
     // update color buffer
     // Note: no depth buffer update
     color[x][y] = over(tri_color, color[x][y]);
  }
}
```

Now we study the **end-to-end rasterization pipeline**:

1. Step 1: Transform triangle vertices into camera space.
2. Step 2: Apply perspective projection transform to transform triangle vertices into normalized coordinate space.
3. Step 3: Discard triangles that lie complete outside the unit cube and clip triangles that extend beyond the unit cube to the cube.
4. Step 4: Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen size).
5. Step 5: Compute triangle edge equations and attribute equations.
6. Step 6: Sample coverage, evaluate Z buffer,
7. Step 7: Compute triangle color at sample point (color interpolation, sample texture map).
8. Step 8: Perform depth test and update depth value at covered sample if necessary.

9. Step 9: Update color buffer

Rendering Equation

We begin by addressing the fundamental question **what is color?** Light is electromagnetic radiation (a form of energy which is all around us) and color is what we interpret as some of the frequency in the electromagnetic spectrum. Light is oscillating electric and magnetic field and the frequency determines the color of light. It is important to point out the difference between frequency (i.e. time difference between two peaks) and wavelength (i.e. space between two peaks). Those concepts are related to speed and in general electromagnetic radiations travel in vacuum at the speed of light. Heat generates light: in facts by combining some physics concept we know that the motion of charges particles generates an EM field and every object moving is hence producing color. The frequency is determined by the temperature. The visible spectrum (the part we refer to as color) is only a small part of the whole spectrum and has a wavelength from 400 (blue) to 700 (red) nanometers. Natural light, i.e. the light which comes from the sun, consists of a mixture of frequency (it is a very hot body and it emits energy in the entire spectrum). Much of the energy which reaches the earth from the sun is in the visible spectrum range and a lot of other frequencies are absorbed by the atmosphere. Why do we care about the sun? Because for us it is useful to compare artificial light (e.g. the one which comes from a bulb) with the light which comes from the sun. In general it is possible, given a source of light, to generate its emission spectrum: the one of natural light brings every color (a little bit more of blue than red, but the pattern is pretty much homogeneous), while for example the cool white LED has only a very tiny fraction of red. On the market there are a lot of different lightbulbs with different spectrum and usually there is a tradeoff between quality of the spectrum and power efficiency. In general we have to consider both the emission spectrum and the absorption spectrum (which is very useful when we talk about paints, ink, ...) and this spectrum tells us how much light is absorbed (i.e. turned into heat) by an object. In general the most important concepts to describe color are:

- Intensity
- Emission
- Absorption

as a function of frequency. Everything else is just a convenient approximation ;)
An interesting example is reflection: if a light source has emission spectrum $f(\lambda)$ and the surface has reflection spectrum $g(\lambda)$, then the resulting intensity is the product $f(\lambda) \cdot g(\lambda)$.
Now we go a step further in our travel into the world of colors and we see **how electromagnetic radiation ended up perceived by a human as a certain color?** The eye is a photosensor which takes light as input (formally, electromagnetic distribution over wavelength $\Phi(\lambda)$) and generate an output response encoded as electrical signal. How the response is generated depends by the spectral response function $f(\lambda)$ which is the sensitivity of sensor to light of a given wavelength. Greater $f(\lambda)$ corresponds to a more efficient sensor, i.e. when $f(\lambda)$ is large a small amount of light at wavelength λ will trigger a large sensor response. The output can be expressed with the following formula:

$$R = \int_{\lambda} \Phi(\lambda) f(\lambda) d\lambda$$

There are two very important structures which take the spectrum and transform it into signals to the brain:

- Rods: light-sensitive but not color sensitive, used under dark conditions, 120 millions in an human eye, spread in different regions of the retina.
- Cones: color sensitive, used under high-light viewing conditions, 6-7 millions in the human eye, three types of cones which are specialized for different frequencies, concentrated in the fovea.

In order to represent colors in a digital way we use models like RGB (which are inspired by the three types of cones). RGB is an additive method, but there are also subtractive methods such as CMYK. Another manner to represent color is the YCbCr which has a channel for the luminance, another one for the blue-yellow deviation from gray and a third one to the red-cyan deviation from gray. This method is very useful to do compression.
Now we change a little bit perspective from the beginning of the lecture and we imagine light as photons who travel in straight lines and hit objects. The brightness of the image is proportional to the energy of photons hitting the object. We note the following important concepts:

- Radiant energy: number of total hits
- Radiant energy density: hits per area
- Radiant flux: number of total hits in one second
- Irradiance: number of hits per second per unit area

A camera has an array which measures the things we just explained and then generates the image.
In order to understand the rendering equation we introduce the concept of radiance $L(p, \omega)$, i.e. the energy per unit time per area per solid angle along a ray defined by origin point p and direction ω. On a surface the incident radiance is not the same as the exitant radiance. This quantity characterizes the distribution of light in an environment and in general rendering is all about computing radiance. Now we see the rendering equation:

$$L_0(p, \omega_0) = L_e(p, \omega_0) + \int_{H^2} f_r(p, \omega_i \to \omega_0) L_i(p, \omega_i) \cos \Theta_i d\omega_i$$

where:

- $L_0(p, \omega_0)$ is the observed radiance at the point of interest in the direction of interest.
- $L_e(p, \omega_0)$ emitted radiance.
- $f_r(p, \omega_i \to \omega_0)$ is the scattering function which includes information about the behaviour of the photon (in facts photons are not always just reflected as in a mirror, but the photon may be reflected in a lot of different ways).
- $L_i(p, \omega_i) \cos \Theta_i$ is the incoming radiance from direction i.

the key challenge about this equation is that it is recursive: to evaluate incoming radiance we have to compute another integral.

Ray tracing

In ray casting we shoot races from each one of the pixels in the image plane through the pinhole. Since two points define a line we can ask ourselves: which 3D geometry are hit by this ray? Basically we shot rays from every pixel and as soon as we hit an object we know how to color a pixel. We have:

- Sample: a ray in 3D
- Coverage: does ray *hit* triangle (ray triangle intersection tests)
- Occlusion: closest intersection along ray

Difference between rasterization and ray casting:

- Rasterization proceeds in triangle order; most processing is based on 2D primitives; has a depth buffer.
- Ray casting: proceeds in screen sample order; no depth buffer; natural order for rendering transparent surfaces; must store entire scene

Note that in general rasterization is more efficient.
Consider the problem of shading, if we use rasterization we first render the depth buffer from the point of view of the light source and then the image from the point of view of the camera. If during the rendering from the camera we encounter an object which is in the depth buffer from the light source than this point will be brighter, otherwise it will be in shadow. With ray tracing we hit an object and then from the hit point we do a ray towards the light source: if we don't hit any other object this point will be bright, otherwise a shadow. In general with recursive ray tracing we keep in account all physics law about light (e.g. Snell's law) and we modify the image by using this information.
In general, in order to do ray tracing (but also other operations in computer graphics), we need to answer several geometric queries:

- Closest point on 2D line: use orthogonal projection.
- Closest point on line segment: find closest point on the line; if in segment then we are over, otherwise return closest end point.
- Intersect ray with implicit surface: parametrize ray, put this into implicit representation, solve for ray parameter.
- Ray triangle intersection: compute ray-plane with triangle intersection, calculate barycentric coordinates of the point, if all coordinates positive then hit, otherwise miss.
- Given a scene defined by set of N primitives and ray r, find closest intersection point: naive is to iterate through all points (very inefficient if we have tons of points). An alternative is to group the objects into groups and then hitting the ray with a simple form (e.g. cube) that contains the group. In this way we do a kind of binary search over the objects.

In order to do the partitioning we have two possibilities:

- **Primitive partitioning (BVH):** partition nodes into disjoints sets (but sets may overlap in space). How to do the partition? Give to each partition the same number of primitives or minimize empty space.
- **Space partitioning (KD tree):** partition space into disjoint regions (but the same primitive bay be contained in multiple regions of space. If the object move primitive partitioning is better.

Intro to Animation

Until now we have seen different stages to create an image (first we started with a simple cube man, then we learned polygonal meshes and finally we talked about materials and lighting). Now we want to do the next step: we don't want static figures but we want to represent objects in motion.
A very important idea in animation is keyframing, i.e. the fact that starting from some images we have to fill the gaps in order to get the illusion of motion. The gaps can be filled manually by an artist or automatically by a computer. The idea is to specify important events only and the computer fills the gaps via interpolation. Note that events don't have to be positions, but could also be color, light intensity, camera configuration, ... Now we dive into interpolation.
Interpolation: the idea is to connect the dots, e.g. with a line. Using piecewise linear interpolation might not be the best idea because we have a very rough and innatural motion. In general, a spline is any piecewise polynomial function. We have n tuples of the form (t_i, f_i) and we want that for all i our interpolation function g has the property that $g(t_i) = f_i$. For all points between t_i and t_{i+1} we want a polynomial of a given degree (e.g. degree 3 for cubic interpolation). Using too high degree polynomials is not a good idea because the Runge's phenomenon can happen. Suppose that we want to use a cubic polynomial to interpolate two end points. We have:

$$p(t) = at^3 + bt^2 + ct + d$$

We have several possible solutions to interpolate the points. A good idea to get the best possible polynomial is to also match the derivatives at the end points.
If we have n points and we want to interpolate them we use a different polynomial for every interval and we want to match the end points and the first two derivatives.
The properties of a good spline are:

- Interpolation: spline passes exactly through data points.
- Continuity: at least twice differentiable everywhere.
- Locality: moving one control point does not affect the whole curve.

Natural spline satisfies the first two conditions but not the locality. There are also other kinds of interpolation functions, e.g. Hermite Spline satisfy interpolation and locality (but not continuity) and B-Splines, which satisfy locality and continuity (but not interpolation).

Physically-based Animation

We are concerned with non static scene and in order to study this topics there are two branches of physics: kinematic (which describes the movement) and dynamics (which explain why objects move in relation to forces). A very important equation in this context is the second of law of Newton (i.e. the famous $\vec{F} = m\vec{a}$). In computer graphics we say that every system has a configuration $q(t)$ and there are forces which act on this system. We can write the Newton's law as:

$$\ddot{q} = \frac{F}{m}$$

Often it is useful to describe systems with many moving parts. For example one can consider a collection of billiard balls, each with position x_i and collect all the balls into a single vector of generalized coordinates. Hence we can think of this vector which contains all the positions of the billiard balls as a single point which moves along a trajectory in a very high dimensional space.
One can also write second order differential equations with a system of first order differential equations, e.g.:

$$\begin{cases} \dot{q} = v \\ \dot{v} = \frac{F}{m} \end{cases}$$

In the physics courses we have seen different problems which can be solved by hand, but in reality it can be very difficult to solve differential equations and hence we need computers and numerical methods to approach this kind of problems.
In order to solve ODEs numerically we recall some concepts of the course *Numerical Methods for CSE*:

- Solving ODEs numerically requires numerical time integration of the form $q(t+h) = q(t) + \int_t^{t+h} \dot{q} dt$, which can be done with discrete approximations of the form $q_{i+1} = q_i + \Delta q_i$. In order to approximate Δq_i we can use:
  - Rectangle rule: $\Delta q_i \approx \dot{q}(t) \cdot h$
  - Midpoint rule: $\Delta q_i \approx \dot{q}(t + \frac{h}{2}) \cdot h$
  - Trapezoid rule: $\Delta q_i \approx \frac{\dot{q}(t)+\dot{q}(t+h)}{2}$

  where those rule differ in accuracy and number of function evaluations.
- Forward Euler: walk a little bit in the direction of the derivative, easy but unstable.
- Backward Euler: instead of using velocity at the current configuration, evaluate velocity at new configuration. This is stable, but involves solving a system of (maybe nonlinear) equations.