Computer Graphics

Ray Casting

- **Ray Casting** estimates the projection of a scene onto a sensor using ray intersections
- Challenge: Incoming and outgoing light at every point of the scene needs to be known (in theory)
-
  - Primary rays solve the visibility Problem (What is visible?)
  - The rendering equation / A Shading Model uses secondary rays solve color, brightness etc.
- Ray Tracing: Function $L(p, \omega_i)$ gives Light at point p coming from direction ω_i
- Outgoing light in direction ω_o can then be calculated as a weighted sum/integral of the incoming rays
- Rays are specified as equation $r(t) = ot + d$

Implicit surfaces

- Defined as set of points (x, y, z) such that $f(x, y, z) = 0$ for a given function f
- Ray intersects an implicit surface if $\exists t[f(r(t)) = 0]$
- For example: A plane in normal form is given by $n \cdot (p - r) = 0$, therefore the intersection with a ray is given by

  $$n \cdot (o + td - r) = 0 \implies t = \frac{(r - o) \cdot n}{n \cdot d}$$

  Note that such an intersection only exists if n ist not orthogonal to d, since otherwise $n \cdot d$ would be 0.
- The surface normal of an implicit function f at a point p is given by the gradient

  $$n_p = \nabla f(p) = \left(\frac{\partial}{\partial x} f(p), \frac{\partial}{\partial y} f(p), \frac{\partial}{\partial z} f(p) \right)$$

Quadrics

- Implicit surfaces given by quadratic equations, which means there can be at most two intersections with a given ray.
- Examples:
  - Sphere: $\frac{x^2+y^2+z^2}{a^2} - 1 = 0$
  - Ellipsoid: $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1 = 0$
  - Paraboloid: $\frac{x^2+y^2}{a^2} - z = 0$
  - Hyperboloid: $\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} - 1 = 0$
  - Cone: $\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 0$
  - Cylinder: $\frac{x^2+y^2}{a^2} - 1 = 0$

Parametric surfaces

- Given by set of equations with two parameters:

$$x = f(u,v), y = g(u,v), z = h(u,v)$$

- Normal vector is given by:

$$n_{u,v} = \left(\frac{\partial}{\partial u}f(u,v), \frac{\partial}{\partial u}g(u,v), \frac{\partial}{\partial u}h(u,v)\right) \times \left(\frac{\partial}{\partial v}f(u,v), \frac{\partial}{\partial v}g(u,v), \frac{\partial}{\partial v}h(u,v)\right)$$

- Examples:
  - Cylinder, with parameters $\phi \in [0, 2\pi)$ and $v \in [0,1]$:

$$x = \cos(\phi),\ y = \sin(\phi),\ z = z_{min} + v(z_{max} - z_{min})$$

  - Sphere, with parameters $\phi \in [0, 2\pi)$ and $\theta \in (0, \pi]$

$$x = \cos(\phi)\sin(\theta),\ y = \sin(\phi)\sin(\theta),\ z = \cos(\theta)$$

  - Discs
  - Cones
- Can be used to render partial objects by restricting the range of the parameters

Combined Objects

- Constructive Solid Geometry (CSG): Combine objects using boolean operators
- Only for closed surfaces with defined volumes
- **Union** $A \cup B$: Closest intersection with either, **Intersection** $A \cap B$: Closest intersection with A that is inside B or vice versa, **Difference** $A\ B$: Closest intersection with A that is not in B

Triangles

- Given parametrically by:

$$p(b_1, b_2) = (1 - b_1 - b_2)p_0 + b_1 p_1 + b_2 p_2$$

  Where p_0, p_1, p_2 are the vertices of the triangle and $b_1 \geq 0$, $b_2 \geq 0$, $b_1 + b_2 \leq 1$

Box Tests

- Intersection test with simple geometry around complex geometry, so that rays that completely miss the mark can be discarded quickly and efficiently
- Generally combined into trees of progressively smaller / more accurate boxes - if ray intersects a box, check intersections with small boxes inside that box

Iso-Surfaces in Grids

- Fluid Particles $\rightarrow$ Density Values on a grid $\rightarrow$ Full Surface

Rendering

- **Colored Light** represented as 3-dimensional Vector with red, green, and blue components
- Surface color is characterized by how much of each color the surface absorbs / reflects
- **Rendering Equation**:

$$L(p, \omega_o) = \int_i \text{mat}(p, \omega_i, \omega_o) L(p, \omega_i) \cos(\theta_i) d\omega_i$$

  So: Outgoing Light is equal to the sum / integral of incoming light from all directions, weighted by the material properties of the material involving light from that direction going into the other direction, times the cosine of the angle between incoming and outgoing light

Shading Models

- Light coming from a point is calculated only considering direct illumination.

$$L(p, \omega_o) = \sum_{i \in I} f(L(p, \omega_i))$$

  Where I ist the set of light sources and f is a function given by the particular rendering model.

- **Lamberts cosine Law**:

$$L(p, \omega_o) = L(p, \omega_i) \cos(\theta_i)$$

- **Phong illumination model:**

$$\begin{aligned} L(p, \omega_o) &= L_{\text{Ambient}} + L_{\text{Diffuse}} + L_{\text{Specular}} \\ &= \alpha \cdot \rho \otimes L_{\text{Indirect}} + \sum_{i \in I} L_i \cdot (n \cdot l_i) \otimes (\beta \cdot \rho + \gamma \cdot \rho_{\text{White}} \cdot (r_i \cdot v)^m) \end{aligned}$$

  Where:
  - $\otimes$ is componentwise multiplication
  - ρ is the color of the surface at the given point
  - L_i is the light given off by light source i
  - l_i is the normalized vector pointing from the given point to light source i
  - n is the surface normal at the given point
  - θ_i is the angle between l_i and n
  - $r_i = 2 \cdot \cos(\theta_i) \cdot n - l_i$, the vector l_i "reflected" along n
  - v is the normalized vector pointing from the given point to the sensor
  - α, β and γ are user-defined scalar coefficients describing the strength of Ambient, Diffuse, and Specular Lighting.
  - m describes the size and brightness of the specular highlight

- The Phong model is limited to opaque surfaces. Reflective or transparent surfaces are not considered.

- The Phong model is not physical, i.e. conservation of energy is generally not respected

Extensions of the Phong Model

- **Considering Distances:** The light at a given surfaces is inversely proportional to the distance to the light source.

$$L_i^{\text{Surf}} = \frac{1}{d^2} L_i \cdot (n \cdot l)$$

- **Fog**: Linear interpolation between computed Light L_{cam} and fog color c_f.

$$L = f(d)L_{\text{cam}} + (1 - f(d))c_f$$

, where d is the distance from the point to the viewer and f is some function describing light weakening as it travels through fog, for example $f(d) = \frac{a-d}{a-b}$ for constants a, b

Other shading models

- Shading models can be evaluated per fragment (i.e. Pixel) or per vertex
- **Flat shading**: Fragments are colored with the color of one specific vertex, very efficient
- **Gouraud shading**: Fragment colors are interpolated from vertext colors. Cheaper than Phong, mach band effect, local highlights may still be lost
- **Phong shading**: Evaluation per Fragment, using normals interpolated from the vertex normals, more costly

Homogeneous Notation

- Allows translation to be expressed as a linear transformation and therefore as a simple matrix multiplication
- The **Model transform** M_m transforms from a model m's Local coordinate system to the global coordinate system
- The **View transform** V transforms from the camera's local coordinate system (with the camera at the origin facing one of the coordinate axes) to the global coordinate system
- The **Inverse View Transform** V^{-1} is applied to all objects in order to translate the entire system into **View Space / Camera Space**, with the camera aligned at the origin as previously mentioned.
- In order to directly translate from local space to view space, the modelview transform $V^{-1}M_m$ is applied. In the case of the camera, this means the modelview transform is simply the identity matrix.

- In **homogeneous Notation**, the position $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$ is represented as (x, y, z, w), with scalar multiples of this vector representing the same position.
- Positions of the form $(x, y, z, 0)$ represent positions at infinity. In practice this is usually used to represent directional vectors as positions.
- Arbitrary affine transformations can now be realized through matrices of the form

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & t_0 \\ m_{10} & m_{11} & m_{12} & t_1 \\ m_{20} & m_{21} & m_{22} & t_2 \\ p_0 & p_1 & p_2 & w \end{pmatrix}$$

  Where m represents ordinary linear transformations (shear, scale, rotation), t represents translation and p represents projection. If p_2 is 0, we have parallel projection (Viewpoint is at infinity), otherwise we have perspective projection. Parallel projection can be orthographic or oblique, perspective projection can involve one vanishing point or two.
- The inverse of a rotation matrix is its transpose.
- Rotation and reflection matrices are orthogonal
- Rotation matrices have determinant 1 and reflection matrices have determinant -1
- Combinations of rotation and translation (i.e. rotations, including rotation around infinity) are known as **Rigid-Body Transforms**
- Scalar multiples of a projection represent the same projection.
- Parallel projections can simply be realized as identity matrices with one of the 1s changed into a 0.
- The **Projection transform** P transform view space (a frustum, with "correct" corner coordinates) into **Clip Space / NDC Space** (a cube, with corners (1,1), (-1,-1) etc, known as the **Canonical view volume**). It is given by

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

  Where the constants refer to the coordinates of the corners (left, right, top, bottom, near, far)

- For $t = -b$ and $l = -r$ this simplifies to

$$P = \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- Can end up looking strange if near plane is too close to zero

Rasterization

- Local Space – $PV^{-1}M_i$ → Canonical view volume → Rasterization → Visibility Tests → Shading → finished image
- Fragment = Candidate for a pixel, Primitive = Line

Vertex Processing

- Input: Vertices
- Output: Vertices
- Computation of Vertex Positions, Color, Texture positions etc.
- Transformation from local space to the canonical view volume ($PV^{-1}M_i$)

Rasterization

- Input: Vertices
- Output: Fragments
- Sets or interpolates fragment attributes using vertex attributes
- Polygon Rasterization: Compute intersections with horizontal scanlines $y = y_i + 0.5$, fill positions between intersections with fragments
- Fragment attributes are interpolated from Vertex attributes

Fragment Processing

- Input: Fragments
- Output: Fragments
- Processes fragment attributes, discards fragments that are not visible
- Texturing, Fog, Antialiasing, ...

- **Scissor Test:** Test if fragment is inside specified rectangle
- **Alpha Test:** Transparency, Billboarding (Sprite that always faces the camera)
- **Stencil Test:** Used for shadows etc.
- **Depth Test:** Are any new fragments behind fragments that are already rendered?

Framebuffer Update

- Get updated framebuffer attributes from processed fragments
- Overwrite old fragment if not transparent, otherwise if alpha ¡ 1 combine old fragment color with new fragment color weighted by the alpha value

Curves

- Defined by parametric functions $x(t), y(t), z(t)$
- For **rational curves**, these functions have the form $\frac{p(t)}{q(t)}$ with p and q being polynomials
- Bezier Curves: Functions are linear interpolations of linear interpolations
- As a result, a degree n bezier curve has the form

$$\sum (1-t)^i t^j p_k$$

for $i+j=n$, with p_k being the control points. For example, for a quadratic bezier curve, the function is:

$$\begin{aligned} q(t) &= (1-t)[(1-t)p_0 + tp_1] + t[(1-t)p_1 + tp_2] \\ &= (1-t)^2 p_0 + 2(1-t)tp_1 + t^2 p_2 \end{aligned}$$

and for a cubic bezier curve, the function is:

$$\begin{aligned} c(t) = (1-t)[(1-t)[(1-t)p_0 + tp_1] + t[(1-t)p_1 + tp_2]] \\ +t[(1-t)[(1-t)p_1 + tp_2] + t[(1-t)p_2 + tp_3]] \\ = (1-t)^3 p_0 + 3(1-t)^2 tp_1 + 3(1-t)t^2 p_2 + t^3 p_3 \end{aligned}$$

- The exact general formula for a degree n bezier curve is:

$$\sum_{i=0}^{n} B_{i,n}(t) p_i$$

Where $B_{i,n}$ are the so-called **Bernstein Polynomials**

$$B_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

In Praxis, curves of degree greater than 3 are only rarely used, because the control points are a lot more difficult to get right.

- **Properties of Bernstein Polynomials**:
  - Partition of Unity:

$$\sum_{i=0}^{n} B_{i,n}(t) = 1$$

  - Positivity:

$$B_{i,n}(t) \geq 0$$

  - Symmetry:

$$B_{i,n}(t) = B_{n-i,n}(1-t)$$

  - Recursive definition:

$$B_{i,n}(t) = (1-t)B_{i,n-1}(t) + tB_{i-1,n-1}(t)$$

  - Endpoint Interpolation:

$$x(0) = p_0, x(1) = p_n$$

  - Endpoint Tangent:

$$\frac{d}{dt}x(0) = n(p_1 - p_0), \frac{d}{dt}x(1) = n(p_n - p_{n-1})$$

- $x(t)$ is part of the smallest convex set that contains all control points (the **convex hull**), i.e. if you draw lines between every control point, the bezier curve is contained fully inside the resulting shape.
- An affine transformation can be applied to a bezier curve by applying it to the control points.
- Every control point affects the entire bezier curve, just weighted at each point by the Bernstein Polynomials.

- General formulation of a **Spline** of degree n:

$$x(t) = GST(t)$$

Curve = Geometry Matrix (Control Points) · Spline Matrix · Basis (Standard Polynomial Basis) For a 2D degree 3 bezier curve with control points $\binom{p_i}{q_i}$, this ends up being:

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} p_0 & p_1 & p_2 & p_3 \\ q_0 & q_1 & q_2 & q_3 \end{pmatrix} \begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

- a 3D cubic bezier curve can be realized by simply replacing the geometry matrix with

$$\begin{pmatrix} p_0 & p_1 & p_2 & p_3 \\ q_0 & q_1 & q_2 & q_3 \\ r_0 & r_1 & r_2 & r_3 \end{pmatrix}$$

- Bezier Curves can be subdivided using the **De Casteljau** algorithm
- A parametric curve is C^k-continuus if the first k derivatives of $x(t), y(t)$ etc. exist and are continuous
- A cubic bezier curve with end points p_i and p_{i+1} and end velocities m_i and m_{i+1} is given by

$$x^i(t) = p_i H_{0,3}(t) + p_i H_{1,3}(t) + m_i H_{2,3}(t) + m_{i+1} H_{3,3}(t)$$

Where $H_{i,n}$ are the cubic hermite basis functions:

$$\begin{aligned} H_{0,3}(t) &= 1 - 3t^2 + 2t^3 = B_{0,3}(t) + B_{1,3}(t) \\ H_{1,3}(t) &= 3t^2 - 2t^3 = B_{2,3}(t) + B_{3,3}(t) \\ H_{2,3}(t) &= t - 2t^2 + t^3 = \frac{1}{3} B_{1,3}(t) \\ H_{3,3}(t) &= -t^2 + t^3 = -\frac{1}{3} B_{2,3}(t) \end{aligned}$$

- **Catmull-Rom Splines** are a variant of Hermite Splines where $m_i = \frac{1}{2}(p_{i+1} - p_{i-1})$ and $m_{i+1} = \frac{1}{2}(p_{i+2} - p_i)$ (So derivatives are given by new, additional control points p_{i-1} and p_{i+2})

Particle Fluids

- **Explicit Euler:**

$$x^{t+h} = x^t + hv^t + O(h^2)$$
$$v^{t+h} = v^t + ha^t + O(h^2)$$

- Alternative Taylor approximation:

$$x^{t+h} = x^t + hv^t + \frac{h^2}{2}a^t + \frac{h^3}{6}\frac{d}{dt}a^t + O(h^4)$$

- **Navier-Stokes-Equation:**

$$a_i^t = -\frac{1}{\rho_i^t}\nabla p_i^t + \nu\nabla^2 v_i^t + \frac{F_i^t}{m_i}$$

  Where:
  - ρ is the density, p is the pressure and ν is the viscosity
  - $-\frac{1}{\rho_i^t}$ represents acceleration due to pressure differences
  - $\nu\nabla^2 v_i^t$ represents Acceleration due to friction between particles of different velocities
  - $\frac{F_i^t}{m_i}$ represents outside forces such as gravity

Smoothed Particle Hydrodynamics (SPH)

- Interpolates quantities at arbitrary positions and approximates derivatives with adjacent particles
- Can be used to compute ρ_i^t, $-\frac{1}{\rho_i^t}\nabla p_i^t$, $\nu\nabla^2 v_i^t$ and $\frac{F_i^t}{m_i}$
- Quantity A_i at an arbitrary position x_i is calculated by sampling known values A_j at positions x_j:

$$A_i = \sum_j A_j \frac{m_j}{\rho_j} W_{ij}$$

  where W_{ij} is a function that weighs contributions of positions based on their distance to x_i, for example a gaussian distribution.

-

$$\nabla A_i = \sum_j A_j \frac{m_j}{\rho_j} \nabla W_{ij}$$

- For example, the density can now be calculated through the **Explicit SPH Form** as:

$$\begin{aligned}\rho_i &= \sum_j \rho_j \frac{m_j}{\rho_j} W_{ij} \\ &= \sum_j m_j W_{ij}\end{aligned}$$

- Pressure calculation:

$$p_i = \max\left(k\left(\frac{\rho_i}{\rho_0} - 1\right), 0\right)$$

  For resting pressure ρ_0 and a user-defined stiffness k

- Pressure acceleration in SPH:

$$\begin{aligned}a_i^p &= -\frac{1}{\rho_i}\nabla p_i \\ &= -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla W_{ij}\end{aligned}$$

- Viscosity acceleration in SPH:

$$\nu\nabla^2 v_i^t = 2\nu \sum_j \frac{m_j}{\rho_j} \frac{v_{ij} \cdot x_{ij}}{x_{ij} \cdot x_{ij} + 0.01h^2} \nabla W_{ij}$$

Neighbor Search

- Particles are stored in a grid
- Edge length equal to the radius of the kernel function
- Compute unique identifier for each particles (for example via space filling curves)
- Sort particles by identifier and store in a hash table

Boundaries

- Boundaries are represented as particles that still contribute to density, pressure and pressure acceleration
- Boundary particles are static fluid particles

Iso-Surface-Reconstruction (Marching Cubes)

- Initialize density values at points on grid using SPH
- $\rho_i < \rho_0 \rightarrow$ Particle outside the fluid, $\rho_i \approxeq \rho_0 \rightarrow$ Particle inside the fluid
- Generate Outline around grid points that are in the fluid using triangle lookup tables

Image Processing

(Chapter 1 and 2 skipped)

Energy Minimization

- Formulate a Task as a minimization problem
- The function $E(x)$ whose minimum you want to find is often called the **Energy Function**, or in machine learning the **Loss Function**
- First step is to formulate assumptions:
  - Similarity to the input image, so minimize

$$E_D(u_{i,j}) := \sum_{i,j}(u_{i,j} - I_{i,j})^2$$

  - Similarity to neighboring values (smoothness), so minimize

$$E_S(u_{i,j}) := \sum_{i,j}(u_{i+1,j} - u_{i,j})^2 + (u_{i,j+1} - u_{i,j})^2$$

- Now solve the problem of minimizing $aE_D(u_{i,j}) + bE_S(u_{i,j})$, where a and b are weighting parameters
- Advantages:
  - Transparency (Clearly stated assumptions)
  - Optimality (Provably optimal under stated assumptions)
  - Easy to analyze aspects of the problem
  - Fewer Parameters than heuristic methods
  - Compatible with other approaches
- Advantages:
  - Metrics can still be somewhat arbitrary - why minimize the squares? (Probabilistic arguments can help here)
  - Choosing weight parameters is not easy (But can be optimized via a validation dataset)
  - Often difficult in practice.
- Actually solving the example problem means solving a very large system of linear equations involving a sparse matrix
- Positive definite Matrices, which we have here, always have an inverse.

- **Convex functions** have a positive second derivative. They always have a unique global minimum and no local minima. Can be minimized by solving $f'(x) = 0$ or through gradient descent.
- Any linear interpolation (= "convex combination") of convex functions is again convex.
- Actually solving the Matrix:
  - **Jacobi Method** - Simple, parallelizable, but converges slowly and cant be computed in-place.
  - **Gauss-Seidel, SOR** - Faster convergence, in place
  - **Conjugate Gradient** - Convergence in a finite number of steps
  - **Multigrid Methods** - Sometimes much faster
- **Jacobi Method**:
  - Decompose A into its diagonal Part D and off-diagonal Part M ($A = D + M$)
  - $Ax = b \Leftrightarrow (D + M)x = b \Leftrightarrow Dx = b - Mx$
  - D can be trivially inverted by inverting every entry
  - Now start with any guess x_0 and iteratively calculate a new value x_{i+1} as:

$$x_{i+1} = D^{-1}(b - Mx_i)$$

  - Iterate until either $Ax - b$ (also known as the residual, r^k) or $(x_{i+1} - x_i)^2$ is small enough
- **Gauss-Seidel Method**:
  - Split M into upper triangle U and lower triangle L
  - Now $Dx = b - Lx - Ux$
  - During iteration, traverse x from top to bottom. This way the new values can immediately be used in the multiplication with L:

$$x_{i+1} = D^{-1}(b - Lx_{i+1} - Ux_i)$$

  - This leads to faster propagation of information and thus faster convergence
  - Converges only if A is positive or negative definite
- **Successive Overrelaxation (SOR):**
  - Add parameter ω to Gauss-Seidel which intentionally overshoots the answer:

$$x_{i+1} = (1 - \omega)x^k + \omega D^{-1}(b - Lx_{i+1} - Ux_i)$$

  - Optimal ω must be determined empirically.

- **Conjugate Gradient (CG):**
  - u and v are conjugate with respect to A if

$$\langle u, v \rangle_A := u^t A v = 0$$

  - Any set of n conjugate vectors $\{p_k\}$ forms a basis of $\mathbb{R}^n$, so the solution x of $Ax = b$ can be written as $x = a_1 p_1 + \ldots + a_n p_n$
  - Then:

$$\begin{aligned} Ax &= \sum_{i=0}^{n} a_i A p_i \\ \implies p_k^\top Ax &= \sum_{i=0}^{n} p_k^\top a_i A p_i = p_k^\top b \\ \implies a_k &= \frac{p_k^\top b}{p_k^\top A p_k} \end{aligned}$$

  - This way an exact solution can be obtained in only n computations.
  - Obtaining the Basis:
    - Start with arbitrary x_0
    - $p_0 = r_0 = b - Ax_0$
    - This is the gradient of

$$E(x) = \frac{1}{2} x^\top A x - b^\top x$$

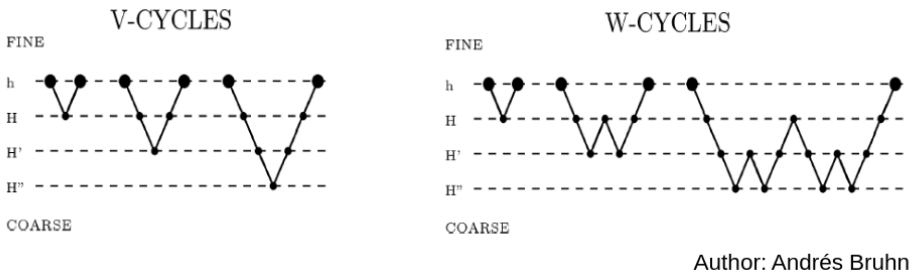
      which is minimized by the solution x.
    - Now iteratively compute:

$$\begin{aligned} a_k &= \frac{r_k^\top r_k}{p_k^\top A p_k} \\ x_{k+1} &= x_k + a_k p_k \\ r_{k+1} &= r_k - a_k A p_k \\ b_k &= \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k} \\ p_{k+1} &= r_{k+1} + b_k p_k \end{aligned}$$

    - Stop when r_k is small, guaranteed solution after n iterations.
  - A must be symmetric and positive definite

  - In image processing, n is the number of pixels, so an exact solution is usually not feasible

- For all of these methods, the speed of convergence depends on the difference between the smallest and largest eigenvalues of A (The **condition number**)
- The condition number can be reduced through pre-conditioners P^{-1} - instead of $Ax = b$, calculate $P^{-1}Ax = P^{-1}b$. The simplest preconditioner is the Jacobi Preconditioner, where $P = D$
- All of these methods have the disadvantage of only acting locally, because of the sparcity of the matrix. This problem can be solved by using a **Multigrid Solver**:
  - A **unidirectional Multigrid** creates downsampled versions of the linear system by downsampling the image and deriving a new system from that.
  - Use one of the previous methods to compute a first approximate solution from the downsampled image
  - Upsample the solution and use this as the initial guess for solving the finer grid
  - Advantages: Solving the coarse image is very fast, simple to implement
  - Disadvantage: The coarse system doesn't always approximate the original system very well
  - **Bidirectional Multigrid**: Downsample the error, not the image



  - Compute first solution at a fine grid:
    - Run some iterations solving $A_h x_h = b_h$ on the fine grid
    - This yields an approximate solution x'_h
    - The remaining error is $e_h x_h - x'_h$
  - Correct error at a coarser grid:
    - Approximate e_h, which is a solution of $A_h e_h = r_h$ ($r = b - Ax'$), by solving the system on the coarser grid ($A_H e_H = r_H$)
    - Obtain better solution $x''_h = x'_h + e_h$

  - Refine solution at a finer grid
    - Run further iterations on the fine grid to decrease remaining error
  - **Full Multigrid**:
    - Start at coarse grid, apply a full W-cycle at each finer level

Motion Estimation

- Goal: Find a vector field $(u, v)(x, y, t)$ that describes the motion of every point at every frame of a video. This video is called **optical flow**.
- Importance of optical flow:
  - Can help with detection, segmentation and labeling of objects (**motion segmentation**)
  - Can be used to estimate 3D motion of objects (**scene flow**)
  - Can help estimate the 3D structure of objects (**structure from motion**)
- Assumption: Neighboring Pixels move the same way
- Otherwise, motion isnt clear - any pixel could have moved to any similarly colored pixel (This is known as the **aperture problem**)
- Another common assumption: Gray value of a moving pixel stays constant:

$$I(x + u, y + v, t + 1) - I(x, y, t) = 0$$

- This constraint is nonlinear, which makes estimation difficult.
- To make the problem easier, the first expression can be linearized through a taylor expansion:

$$I(x + u, y + v.t + 1) = I(x, y, t) + I_x u + I_y v + I_t + O(u^2, v^2)$$

  Where $I_x = \frac{d}{dx} I(x, y, t)$
- This leads to the so-called **linearized optic flow constraint:**

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

  This is precise if the images are smooth and the flow is small.
- The aperture problem is already visible here - there is one equation, but two unknowns. Therefore, it is impossible to find a unique flow vector.
- A second assumption is needed.

The Lucas-Kanade Method

- The **Lucas-Kanade method** assumes constant motion within a local neighborhood. This leads to the following system of equations:

$$I_x(x', y', t)u + I_y(x' + y', t)v + I_t(x', y', t) = 0 \forall x' \forall y' \in \mathcal{R}(x, y)$$

  (for all x', y' in a neighborhood around (x, y))

- In general, this system is over-determined and doesn't have a solution. Instead, we try to minimize the squared error:

$$\text{argmin}_{u,v} \sum_{(x,y)\in\mathcal{R}} (I_x(x, y)u + I_y(x, y)v + I_t(x, y))^2$$

- Necessary conditions for a minimum are:

$$\frac{\partial}{\partial u}E = 2 \sum_{(x,y)\in\mathcal{R}} (I_x(x, y)u + I_y(x, y)v + I_t(x, y))I_x(x, y) = 0$$
$$\frac{\partial}{\partial v}E = 2 \sum_{(x,y)\in\mathcal{R}} (I_x(x, y)u + I_y(x, y)v + I_t(x, y))I_y(x, y) = 0$$

- This leads to the following linear system at each pixel:

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

- Can also use an arbitrary kernel function (usually a gaussian) instead of the previous uniformly weighted box. In this case, the sums come down to a convolution:

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

- A unique solution can only be obtained if the neighborhood contains non-parallel gradients
- Advantage: fast and simple
- Drawbacks:
  - locally constant motion is often not realistic
  - no constraints on the smoothness of the flow field
- The Lukas Kanade method is a **parametric / local method**
- Global methods assume global smoothness, i.e. global dependencies between points
- Generally derived using variational methods, the most basic of which is the **Horn-Schuck** method

The Horn-Schuck Method

- First assumption is the same gray value assumption as in Lucas-Kanade:

$$I_x u + I_y v + I_z = 0$$

- Second assumption: Global smoothness of the flow field:

$$|\nabla u|^2 + |\nabla v|^2 \to \min$$

- Both assumptions can be combined into the following **Energy Functional**:

$$E_(u, v) = \int_\Omega$$