Scientific Visualisation Summary
FS 2013

Pascal Spörri
pascal@spoerri.io

August 3, 2013

This summary is based on the course slides of the Scientific Visualisation course at ETH Zürich[1] from spring semester 2013.

[1] http://www.scivis.ethz.ch/education/scivis_course/notes

Contents

1 Introduction

SciVis is interdisciplinary the fields of application include engineering, natural sciences and medical sciences. There's a common application to all fields: There are *numerical datasets* providing an abstraction from the particular application. The characteristics of such datasets include:

Dimension of domain: Number of coordinates or parameters

Dimension of values: Scalar, vector or tensor fields

Type of data: Discrete values versus discretised data

Type of discretisation: (Un-)structured grid, scattered data

Time dependencies: Static versus time-dependent.

1.0.1 SciVis and InfoVis

Scientific Visualisation is mostly concerned with

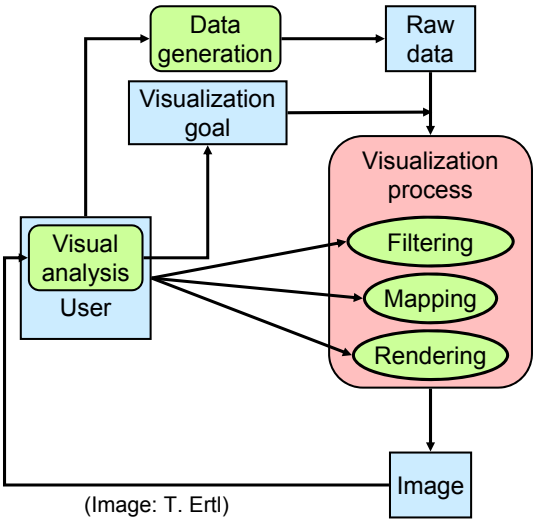
- 2,3,4 dimension spatial or spatio-temporal data
- discretised data

Information Visualisation focuses on:

- High-dimensional, abstract data
- Discrete data
- Financial, statistical, etc.
- Visualisation of large trees, networks, graphs
- Data mining:
  - Finding patterns
  - Clusters
  - Voids
  - Outliers

1.1 Visualisation Scenarios

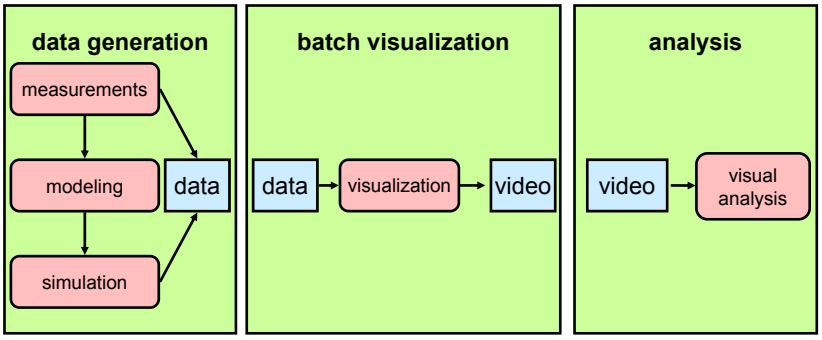
The reference model for visualisation:



(Image: T. Ertl)

1.1.1 Video/Movie

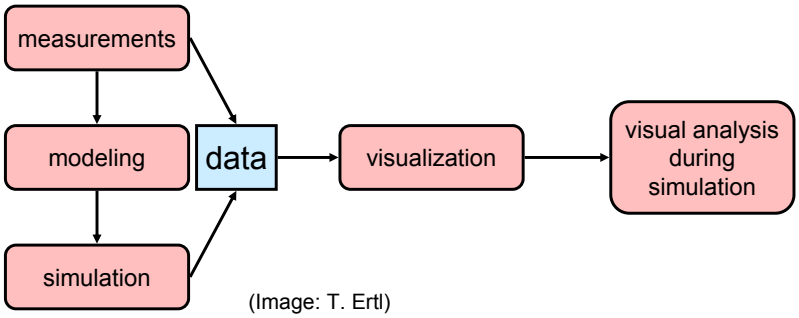
In a first step the data is generated. Then the data is visualised during a batch visualisation step and in the end the video is analized.



(Image: T. Ertl)

1.1.2 Tracking

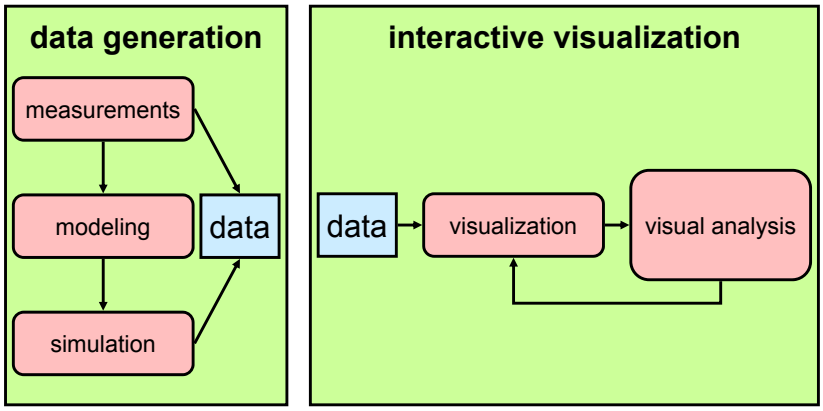
The gathered data is directly visualised and analysed.



(Image: T. Ertl)

1.1.3 Interactive Post Processing/Visualisation

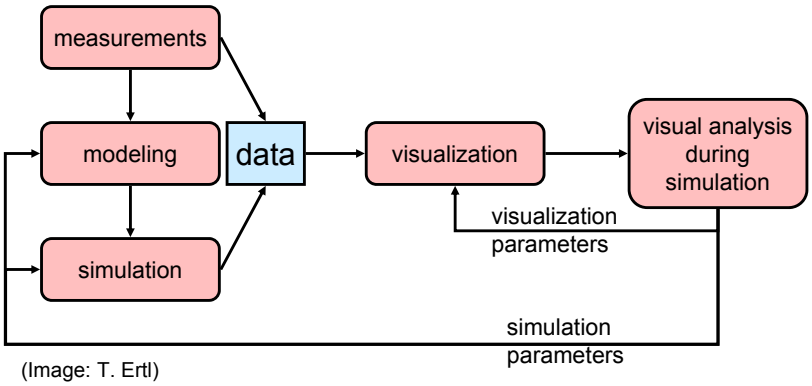
The data generation step is split from the visualisation step.



(Image: T. Ertl)

1.1.4 Interactive Steering/Computational Steering

The visualisation has a direct impact on the simulation and the visualisation.



(Image: T. Ertl)

1.2 Data Discretisations

Types of data sources have typical types of discretisations:

Measurment Data Typically scattered ("mesh-less", no grid)

Numerical Simulation Data

- Structured, block-structured or unstructured grids
- Adaptively refined meshes
- Multi-Zone grids with relative motion
- ...

Imaging Methods Uniform grids

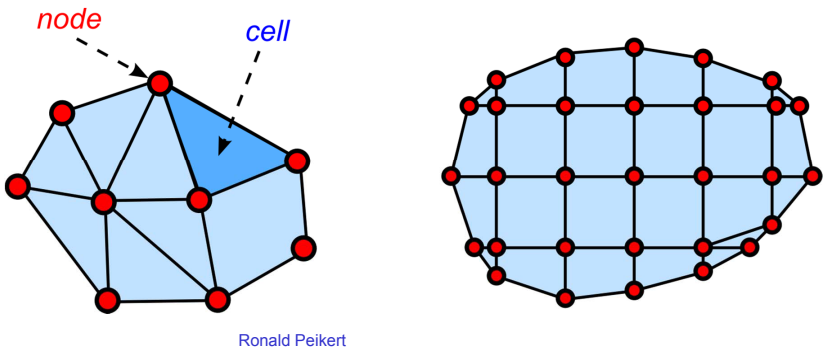
Mathematical Functions (and functionally represented data) can be sampled by demand:

- Uniform
- Adaptive

1.3 Unstructured Grids

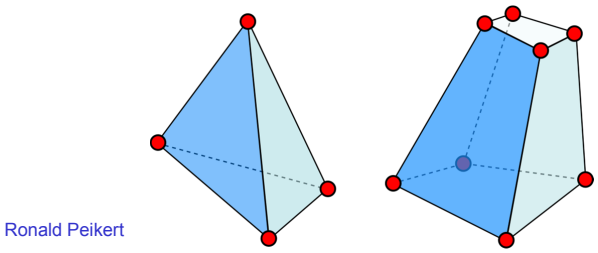
1.3.1 2D Unstructured Grids

Cells are *triangles* and/por quadrangles. The domain can be a surface embedded in 3-space (distinguish n-dimensional from n-space).

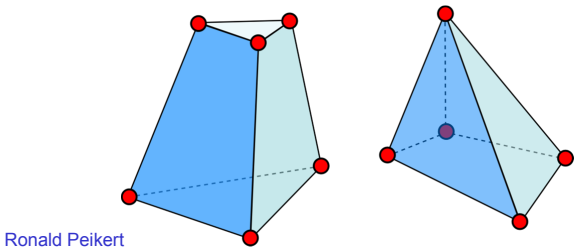


1.3.2 3D Unstructured Grids

Cells are *tetrahedra* or *hexahedra*.



Mixed grids ("zoo meshes") require additional types: *wedge* (3 sided prism), and a *pyramid* (4-sided).



1.4 Structured Grids

Curvilinear Grid (general case) Nodes are given in an array $N_i \times N_j \times N_k$ and the cells are implicit.

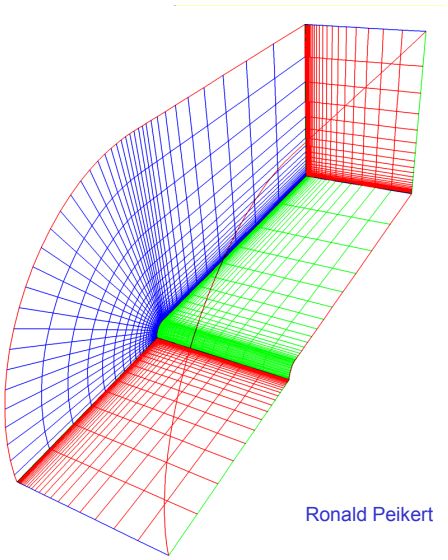


Figure 1: Curvilinear Grid

Rectilinear Grid (special case) The coordinate functions are simpler:

$$x = x(i) \qquad y = y(j) \qquad z = z(k)$$

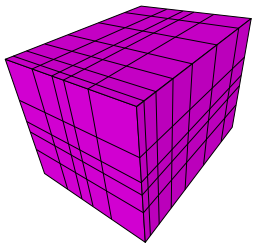


Figure 2: Rectilinear Grid, Source: Wikipedia

Uniform Grid (more special) The coordinates are defined by an *axis-aligned*bounding box.

1.4.1 Point-Sampled Data/Scattered Data

Point sampled data returns only nodes and no cells. Typical data sources are measurement data for example meteorological data.
Options for visualisation include:

Point-Based Methods (relatively few algorithms)

Triangulation for example constrained Delaunay (difficult in 3D)

Resampling onto uniform grid.

1.5 Elementary Visualisation Methods

Scalar Fields can be visualised by plotting its *function graphs*:

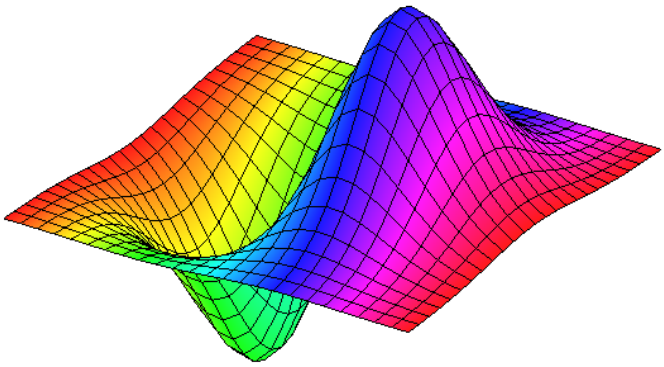
1D Field: The Graph is a curve:

$$y = f(x)$$

2D Field: The Graph is a *height field*:

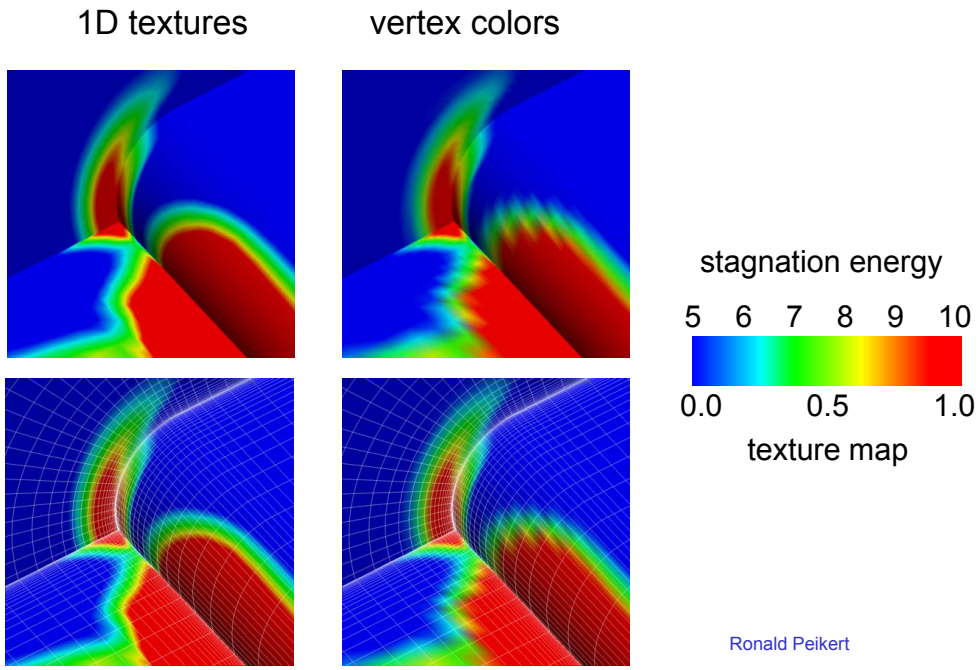
$$z = f(x, y)$$

Visualisation is easy for rectilinear grids:
Painter's algorithm (hidden surface removal in software): Draw cells row by row from back to front.



Ronald Peikert

Scalar fields can also be visualised using *color coding* using *1D texture mapping*. Don't use *vertex colors* and Gouraud shading!



- Problem of RGB colouring mode: The Interpolation is in the wrong colour space (RGB vs. colour table).
- Problem of Colour Index mode: Lighting is not possible.

2 Contouring and Isosurfaces

2.1 Contours

Contours are a set of points where the scalar field s has a given value c:

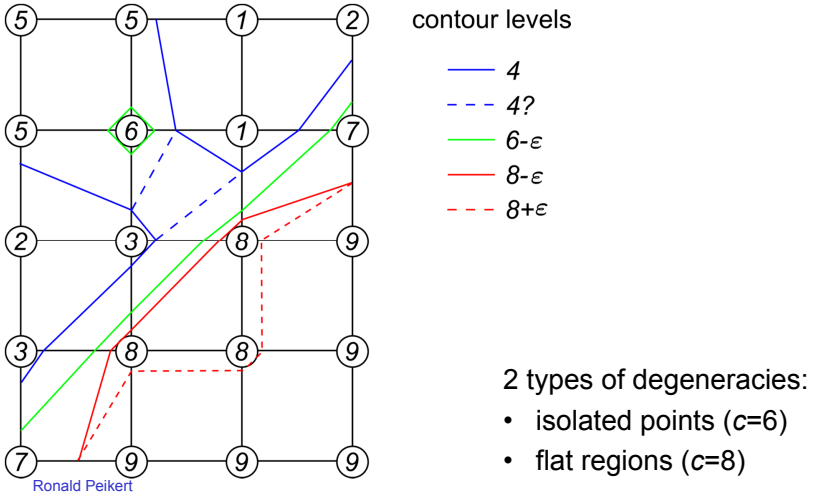
$$\{x \in \mathbb{R}^n : s(x) = c\}$$

Examples in 2D:

- Height contours on maps
- Isobars on weathermaps

Contouring algorithm:

- Find intersection with grid edges
- Connect points in each cell



Topological consistency To avoid degeneracies, use *symbolic perturbations*:

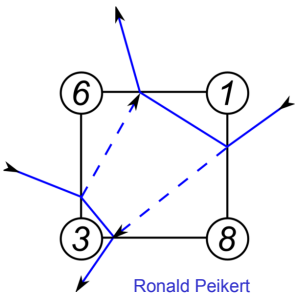
If level c is found as a node value, set the level to $c + \varepsilon$ where ε is an infinitesimal, i.e., $\varepsilon > 0$ and $\varepsilon < x \ \forall x \in \mathbb{R}$.

Then:

- Contours intersect edges at some (possibly infinitesimal) distance from end points.
- Flat regions can be visualised by a pair of contours at $c - \varepsilon$ and $c + \varepsilon$.
- Contours are *topologically consistent*, meaning:

  Contours are *closed*, *orientable*, *nonintersecting lines*.

Ambiguities of contours What is the *correct* contour of $c = 4$?



Two possibilities (from which both are orientable):

- Connecting the high values ——————
- Connecting the low values – – – – – –

2.1.1 Contours in a quadrangle cell

Local Coordinates $(0,0)$ $(1,0)$ $(0,1)$ $(1,1)$

Function Values s_{00} s_{10} s_{01} s_{11}

Bilinear Interpolant

$$\begin{aligned} s(x,y) &= (1-x)(1-y)\ s_{00} + x(1-y)\ s_{10} + (1-x)y\ s_{01} + xy\ s_{11} \\ &= Axy + Bx + Cy + D \end{aligned}$$

with
$$\begin{aligned} A &= s_{11} - s_{01} - s_{10} - s_{00} \\ B &= s_{10} - s_{00} \\ C &= s_{01} - s_{00} \\ D &= s_{00} \end{aligned}$$

- If $A = 0$, then the contour equation is $c = Bx + Cy + D$ and the contours are *straight lines*, all parallel.
- If $A \neq 0$, then the contour equation is

$$c = A\left(x + \frac{C}{A}\right)(y + \frac{B}{A}) + D - \frac{BC}{A}$$

and the contours are *hyperbola* except for the level

$$c = D - \frac{BC}{A}.$$

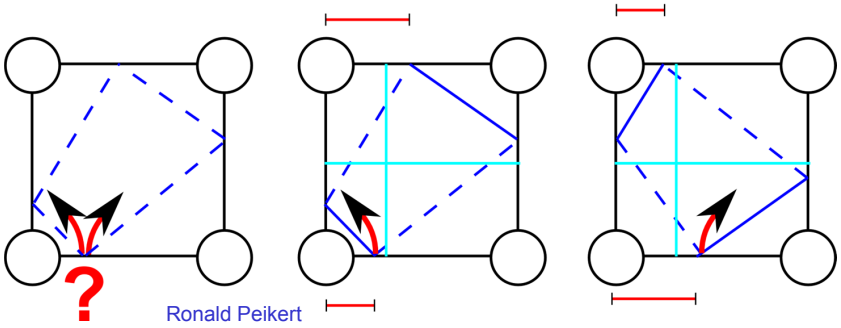
For the special level the contour equation is

$$0 = A\left(x + \frac{C}{A}\right)\left(y + \frac{B}{A}\right),$$

and the contour is a pair of axis-aligned straight lines with

$$x = -\frac{C}{A},$$
$$y = -\frac{B}{A}.$$

Decision can be made without computing special level or saddle points by just comparing fractions of edges:



By using local coordinates, this works also for curvilinear and unstructured grids.
Not that drawing hyperbola instead of straight lines does not lead to better contours: The piecewise bilinear function is not in C^1.

2.1.2 Basic contouring algorithms

Cell-By-Cell algorithms: Simple structure, but generate disconnected segments and require post-processing.

Contour Propagation methods: Complicated, but generate connected contours.

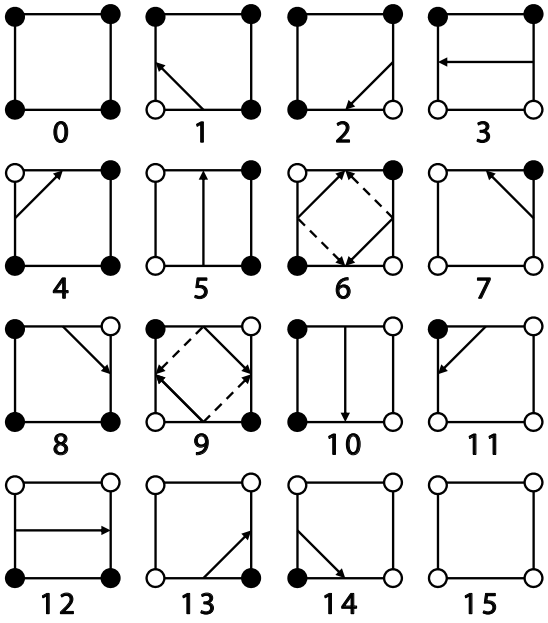
Marching Squares algorithm: Systematic cell-by-cell algorithm. (See below)

2.1.3 Marching Squares

Process nodes in ccw (counter-clockwise order), denoted here as x_0, x_1, x_2 and x_3. At each node x_i compute the reduced field

$$\tilde{s}(x_i) = s(x_i) - (c - \varepsilon) \qquad \text{(which is forced to be nonzero).}$$

Take it's signe as the i^{th} bit of a 4-bit integer. Use this as an index for the lookup table containing the connectivity information.



- $\tilde{s}(x_i) < 0$
- $\tilde{s}(x_i) > 0$

Alternating signs exist in cases 6 and 9. Chose the solid or the dashed line based on topological *consitency*.

Contours in triangle/tetrahedral cells Linear interpolation of cells implies piece-wise linear contours. Since contours are unambiguous let us introduce a "marching triangles" method. This however introduces periodic artefacts.

2.2 The Marching Cubes Algorithm

Contours of 3D scalar fields are known as *isosurfaces*. Before 1987, isosurfaces were computed as contours on planar *slices*, followed by "contour stitching".
The *marching cubes* algorithm computes contours *directly in 3D*:

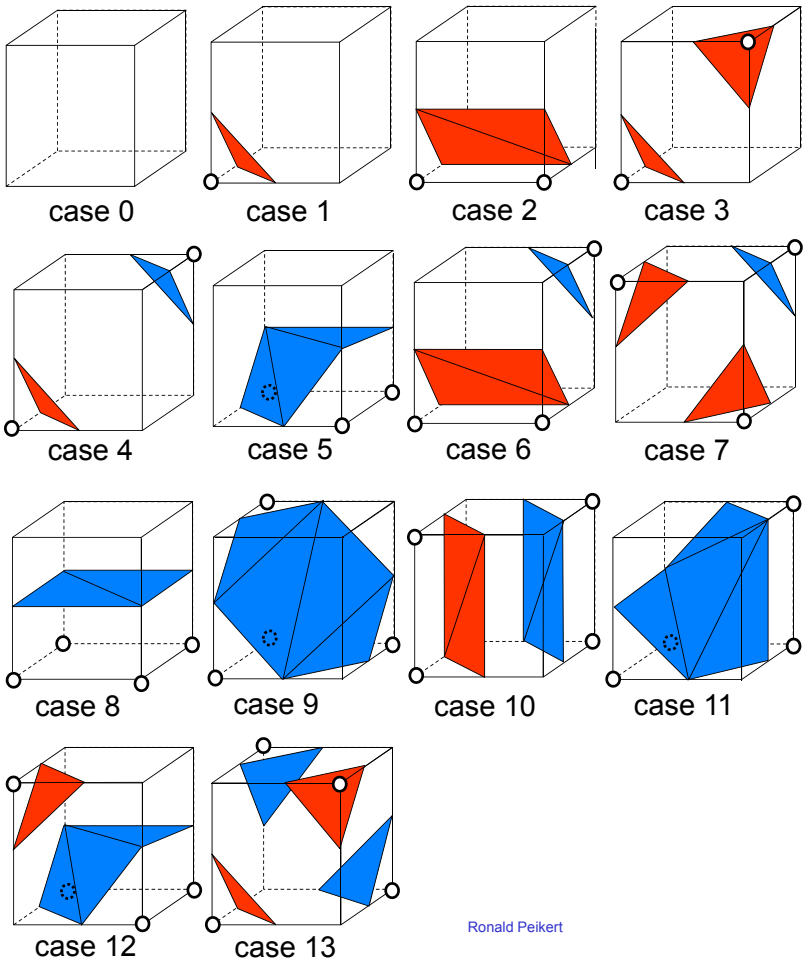
- Pieces of the isosurfaces are generated on a cell-by-cell basis.
- Similar to marching squares, an 8-bit number is computed from the 8 signs of $\tilde{s}(x_i)$ on the corners of a hexahedral cell.
- The isosurface piece is looked up in a table with 256 entries.

How to build up the table of 256 cases?
Lorensen and Cline (1987) exploited 3 types of symmetries:

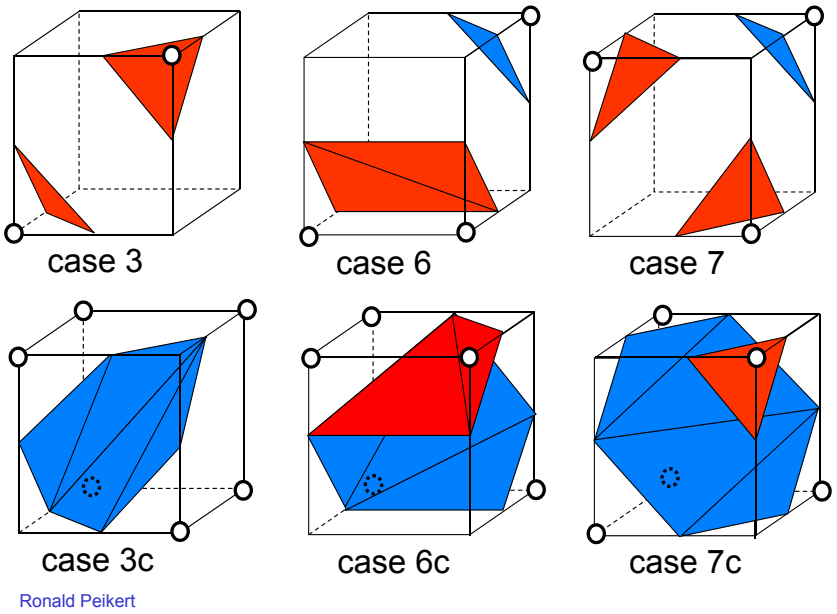
- Rotational symmetries of the cube
- Reflective symmetries of the cube
- Sign changes of $\tilde{s}(x)$.

They published a reduced set of 14 cases:

- White circle indicate positive signs of $\tilde{s}(x)$,
- The positive side of the isosurface is drawn in red, the negative side in blue.



Unfortunately not all pieces fit together (same problem as with marching squares) and additional cases need to be introduced.



The remaining complementary cases are simply obtained by changing the the orientation. Based on these 28 cases, the full 256 cases are obtained by rotations of the cube.

Summary of the algorithm:

1. Pre-processing:
   - Build a table of the 28 cases.
   - Derive a table of the 256 cases, containing info on
     - Intersected cell edges
     - Triangles based on these points
2. Loop over cells:
   - Find sign of $\tilde{s}(x)$ for the 8 corner nodes, giving 8-bit integer.
   - Use as index into lookup table.
   - Find intersection points on edges listed in table, use linear interpolation.
   - Generated triangles according to table.
3. Post-processing steps:
   - Connect triangles (share vertices)
   - Compute normal vectors:
     - By averaging triangle normals (problem: thin triangles!).
     - By estimating the gradient of the field $s(x)$ (better).

2.3 The Asymptotic Decider Algorithm

Motivation for a different isosurface algorithm: Marching cubes can produce "bad" topology.
Asymptotic decider algorithm (Nielson and Hamann 1991):

- Generate topologically *correct* contours (as oriented straight line segments) on the cell surfaces.
- Connect these around the cell, resulting in one or more polygons.
- Triangulate the polygons.

In general, the AD algorithm generates better isosurfaces. However,

- It cannot be easily implemented with a table like Marching Cubes (too many cases).
- It generates polygons with up to 12 sides (Marching Cubes: up to 7).
- The topology is correct with respect to the trilinear interpolant, but the geometry can deviate.
- Some polygons cannot be "cleanly" triangulated.

2.4 The Dividing Cubes Algorithm

An early *point-based* algorithm (Crawford et al. '87): For each cell

- Check whether it is intersected by the isosurface:

$$\min_{i \in \text{cell}} s_i < c < \max_{i \in \text{cell}} s_i$$

- Subdivide the intersected cell into $m \times m \times m$ subcells using trilinear interpolation.
- Draw the centers of all intersected subcells.
- To light the points estimate the gradient and use it as the normal vector.

2.5 Optimised Isosurface Algorithms

Approaches to speeding up isosurface computation:

View Dependent algorithms:

- Occluded triangles are not computed
- GPU-based isosurface computation and redenring

Data Processing for fast computation of *multiple* isosurfaces (multiple levels), for example for interactive exploration of the data.

- Many methods: Octree, Extrema Graph, Span Space.
- Common Goal: Avoid computation in non-intersected cells.

2.5.1 The Span-Space Algorithm

Method by Livnat (1996).

1. Pre-Processing. For each element
   - Compute min and max ,
   - Treat $(\min, \max)$ as a point in the *span space* (Euclidean plane).
   - Store points in boxes, non-empty boxes are stored in a linked list.
2. Computation of the isosurface level c:
   Find the intersected cells in the *quadrant* $\min < c$, $\max > c$.

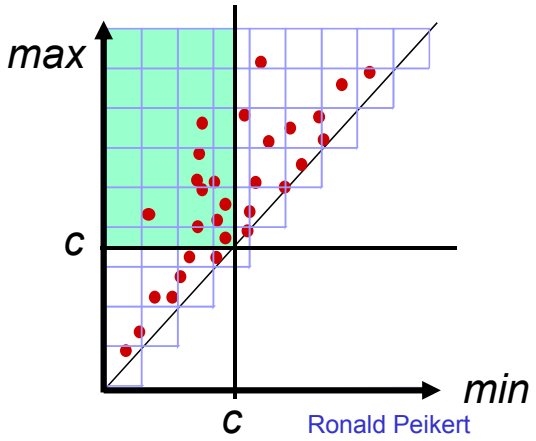
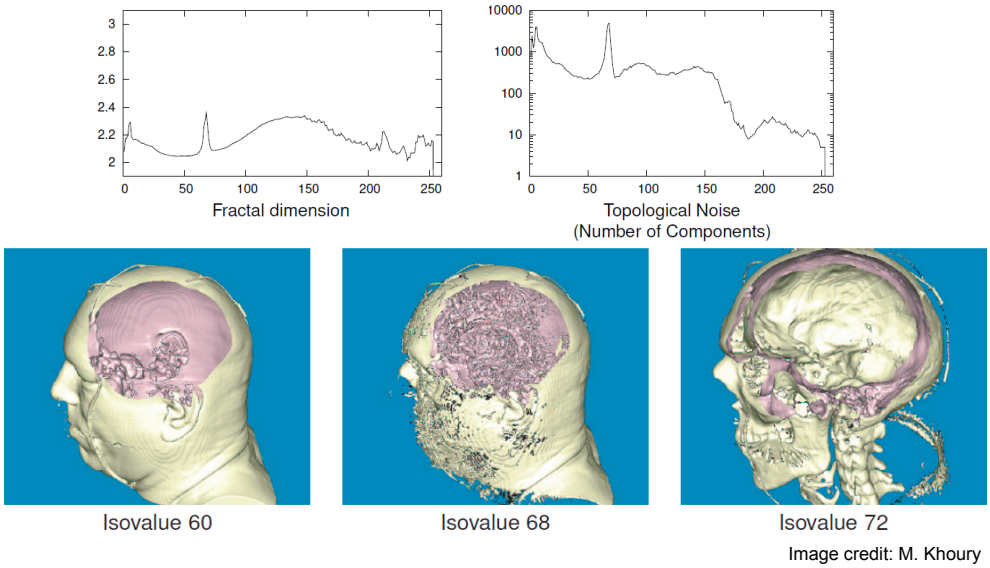


Figure 3: Node distribution in span space.

This algorithm yields a performance gain for datasets with a small local variation, i.e. points in the span space are distributed on the diagonal.

2.6 Selecting Contour Levels

Several types of isosurface statistics can help with level selection.



2.7 Limitations of Isosurfaces

Isosurfaces represent only a single level within the data range. In practical data, there is often not a single "interesting" level.
Transparent rendering of multiple isosurfaces is possible, but

- Limited to a small number of surfaces by visibility
- Alpha-blending require depth sorting.

Alternatives:

- *Feature Extraction methods*, e.g. detecting "blobs" (maximal ellipse-like contours)
- *Volume rendering* can show ranges of "interesting levels of the field and/or its gradient.

3 Raycasting

3.1 Direct Volume Rendering

Volume rendering (sometimes called *direct volume rendering*) stands for methods that generate images directly from 3D scalar data. "Directly" means: *no intermediate geometry* (such as an isosurface) is generated.
Volume rendering techniques

- Depend strongly on the grid type.
- Exist for structured and unstructured grids.
- Are predominantly applied to uniform grids with 2D or 3D image data with *cell-centered* data. Cell-centered data
  - are attributed to cells (pixels, voxels) rather than nodes,
  - can also occur in (finite volume) CFD datasets,
  - are converted to node data:
    - By taking the *dual grid* (easy for uniform grids: n cells $\rightarrow n-1$ cells!),
    - or by interpolating.

3.2 Raycasting

Raycasting is historically the first volume rendering technique. It is very similar to *raytracing*:

- *image-space* method: The main loop iterates over the pixels of the output image,
- a *view ray* per pixel (or per subpixel) is traced backward,
- and samples are taken along the ray and *composited* to a single color.

The differences are

- No secondary (reflected, shadow) rays,
- the transmitted ray is not refracted,
- more elaborate compositing functions,
- and samples are taken at intervals (not at object intersections).

3.2.1 Compositing

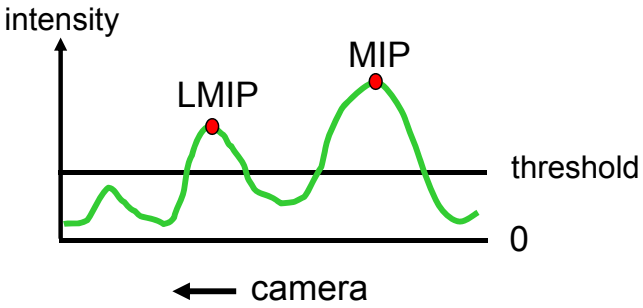
Two simple compositing functions can be used for previewing:

Maximum intensity projection (MIP):

- Maximum of sampled values
- Result resembles X-ray image.

Local maximum Intensity projection (LMIP):

- Choose first local maximum which is above a prescribed threshold.
- Process approximates occlusion.
- It is faster and better than MIP.



3.2.2 α-compositing

Assume that each sample on a view ray has a *color* and *opacity*.

$$(C_0, \alpha_0), \ldots, (C_N, \alpha_N) \qquad C_i \in [0,1]^3, \quad \alpha_i \in [0,1]$$

where the 0^{th} sample is next to the camera and the N^{th} one is a (fully opaque) background sample:

$$C_N = (r, g, b)_{\text{background}}$$
$$\alpha_N = 1.$$

α-compositing can be defined recursively:
Let C_f^b denot the *composite color* of samples f, $f+1$, ..., b. The recursion formula for *back-to-front* compositing yields:

$$C_b^b = \alpha_b C_b$$
$$C_f^b = \alpha_f C_f + (1 - \alpha_f) C_{f+1}^b.$$

With *transparency* set to $T_i = 1 - \alpha_i$ we get a *closed formula* for α-compositing:

$$C_f^b = \sum_{i=f}^{b} \alpha_i C_i \prod_{j=f}^{j-1} T_j$$

The *front-to-back* compositing can be derived from the closed formula: Let T_f^b denote the *composite transparency* of samples f, $f+1$, ..., b:

$$T_f^b = \prod_{j=f}^{b} T_j.$$

Then the *simultaneous recursion* for front-to-back composition is:

$$\begin{aligned} C_f^f &= \alpha_f C_f \\ T_f^f &= 1 - \alpha_f \\ C_f^{b+1} &= C_f^b + \alpha_{b+1} C_{b+1} T_f^b \\ T_f^{b+1} &= (1 - \alpha_{b+1}) T_f^b. \end{aligned}$$

The emission-absorption model How realistic is α compositing? The *emission-absorption* model (Sabella 1988) yields a basic *volume rendering equation*"

$$L(x) = \int_x^{x_b} \varepsilon(x') \exp\left(- \int_x^{x'} \tau(x'') dx'' \right) dx'$$

The equation describes the *radiance* arriving along a ray at the position x on this ray. The *emission* function $\varepsilon(x)$ describes the photons "emitted" by the volume along the ray. The *absorbtion* function $\tau(x)$ is the probability that that photon traveling over a unit distance is lost by absorption.
The emission-absorption model is based on the Boltzmann transport equation in statistical physics, but completely *ignores scattering*. In more general models $\tau(x)$ is an *extinction function* having both an absorption term and a scattering term.
Instead, in the emission-absorption model:

- *Incident scattering* is modelled by the emission function
- *Loss by scattering* can be thought to be part of the absorption.

The discrete version of the emission absorption model:

$$L(x) = \sum_{i=0}^{n} \varepsilon_i \Delta x \exp\left(- \sum_{j=0}^{i-1} \tau_j \Delta x \right) = \sum_{i=0}^{n} \varepsilon_i \Delta x \prod_{j=0}^{i-1} e^{-\tau_j \Delta x},$$

matches the α-compositing formula

$$C_f^b = \sum_{i=f}^{b} \alpha_i C_i \prod_{j=f}^{j-1} (1 - \alpha_i)$$

and gives interpretations of "opacity" and "color":

$$\begin{aligned} \alpha_i &= 1 - e^{\tau_j \Delta x} \approx \tau_j \Delta x && \text{if } \Delta x << 1 \\ \alpha_i C_i &= \varepsilon_i \Delta x. \end{aligned}$$

The product $\tilde{C} = \alpha_i C_i$ is called a *premultiplied* or *associated* colour.

3.3 Transfer Functions

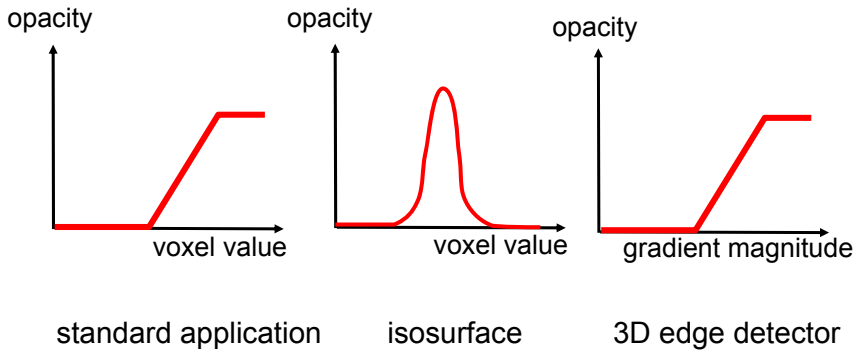
Transfer functions map raw voxel data to opacities and colours as needed for the α-compositing. Inputs of the TF (one or more):

- Voxel value $s(x)$
- Gradient magnitude $\|\nabla s(x)\|$
- Higher derivatives of $s(x)$.

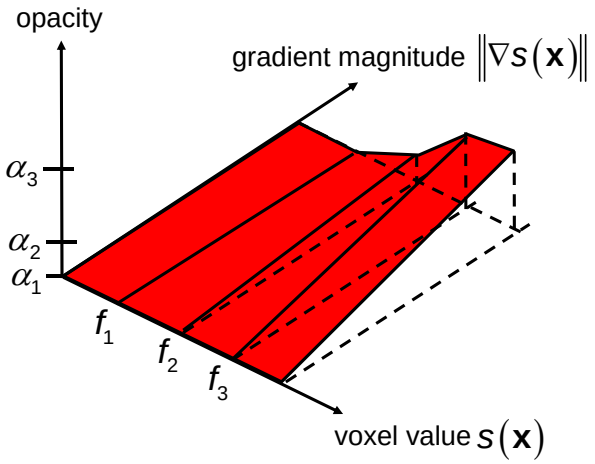
Opacity transfer function $\alpha(s(x), \|\nabla s(x)\|, \ldots)$

Color transfer function $C(s(x), \|\nabla s(x)\|, \ldots)$

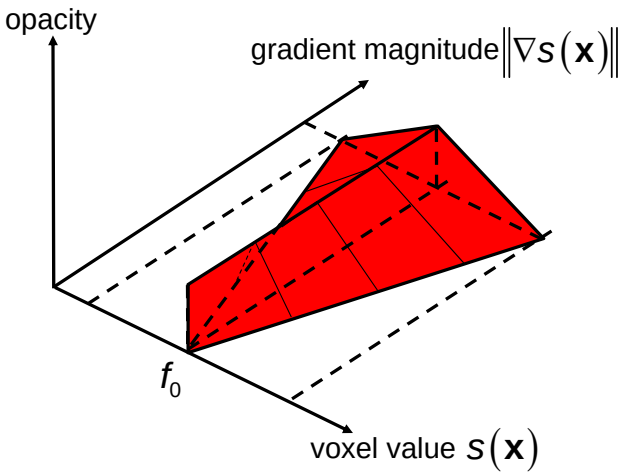
In general TF don't depend on *spatial location*, exception for *focus and context* techniques.
By choosing different opacity transfer functions different types of applications can be achieved.



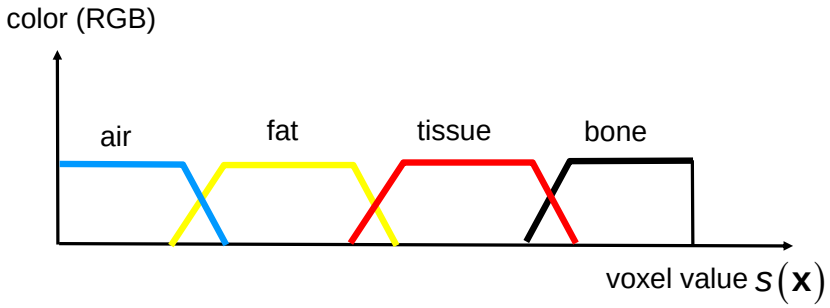
Example of a *bivariate* (=2D) transfer function:



Example of a bivariate transfer function for an isosurface of constant thickness.



The color transfer function allows to make a simple *classification*.



Pre-classification In pre-classification, the voxels can also be lit:

- The gradient is perpendicular to the local isosurface. It can be used as a normal vector for *Phong lighting* (without rendering the isosurface itself).
- *Reflection coefficients* can be assigned by a separate transfer function ("materials" instead of only colors).
- *Diffuse lighting* can be applied to the entire volume dataset as a pre-processing since it's independent of the viewing direction.

3.3.1 Pre- vs. Post-classification

For quality reasons, current volume rendering implementations often use *post-classficiation*.

Pre-Classification

1. Transfer functions are applied to voxels.
2. Results are interpolated to sample locations.

Post-Classification

1. Raw data are interpolated to sample locations.
2. Transfer functions are applied to sampled data.

3.4 Preintegration

Idea (Engel 2001): Simulate *infinitely many* interpolated samples between two successive samples $s_i = s(x_i)$ and $s_{i+1} = s(x_{i+1})$. Assuming that

- Field $s(x)$ varies *linearly* between samples
- and that the transfer functions don't depend on derivatives.

The discrete formula for opacity at a sample was

$$\alpha_i = 1 - e^{-\tau_i \Delta x}.$$

The continuous version for a sample interval $[x_i, x_{i+1}]$ is

$$\alpha_i = 1 - e^{-\int_{x_i}^{x_{i+1}} \tau(s(x))dx}.$$

Assuming now $s(x)$ to be linear between samples, we get:

$$\alpha_i = 1 - \exp\left(-\frac{d}{s_{i+1} - s_i} \int_{s_i}^{s_{i+1}} \tau(s)ds\right) \qquad \text{with } d = \|x_{i+1} - x_i\|,$$

which is called a *preintegrated opacity transfer function*.
The integral

$$\int_{s_i}^{s_{i+1}} \tau(s)ds = \int_0^{s_{i+1}} \tau(s)ds - \int_0^{s_i} \tau(s)ds$$

can be evaluated by two lookups in a precomputed table of

$$\int_0^s \tau(s')ds'.$$

The composite colour of the same interval

$$C_i = \int_{x_i}^{x_{i+1}} \varepsilon(s(x)) \exp\left(-\int_{x_i}^{x} \tau(s(x'))dx'\right) dx,$$

simplifies for linear $s(x)$ to:

$$C_i = \frac{d}{s_{i+1} - s_i} \int_{s_i}^{s_{i+1}} \varepsilon(s) \exp\left(-\frac{d}{s_{i+1} - s_i} \int_{s_i}^{x} \tau(s')ds'\right) ds,$$

which can be precomputed for all combinations of s_i, s_{i+1} and d.

3.4.1 Extinction-based volume rendering

Instead of an opacity Transfer function, use an extinction TF [Schlegel 2011]. Advantage: Extinction is *additive*.

- Riemann sums for numerical integration give better accuracy.
- Additivity can be used for efficient lighting.

Screen-Space ambient occlusion Approximate the fraction of ambient light that is occluded. Method: Compute the total extinction per shell (boxes approximating spheres) b using a *summed area table*.

4 Object Space Volume Rendering

In object space rendering methods the main loop is not over the pixels but over the objects in 3-space. In case of direct volume rendering "objects" can mean:

- Layers of voxels: Use *Image compositing* methods:
  - 2D texture based
  - 3D texture based
- Voxels: Use *splatting* methods.
- Cells: Use *cell projection* methods.

4.1 Texture-based volume rendering

4.1.1 Volume rendering with 2D texturemapping

- Use planes parallel to the *base plane*. The base plane is the front face of the volume which is "most orthogonal" to the view ray.
- Draw the texture rectangles using a bilinear interpolation filter.
- Render back-to-front using α-blending for the α-composition.

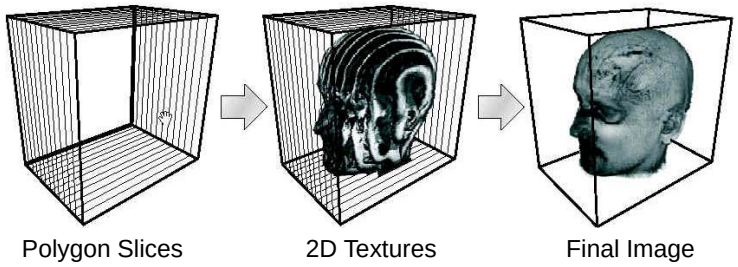


Image credit: H.W.Shen, Ohio State U.

4.1.2 Volume rendering with 3D texture mapping

Cabral 1994.

- Use the voxel data as the 3D texture.
- Render an arbitrary number of slices (eg. 100 or 1000) parallel to the image plane (3- to 6- sided polygons).
- Back-to-front compositing as in the 2D texture method.

This method is limited by the size of the texture memory.

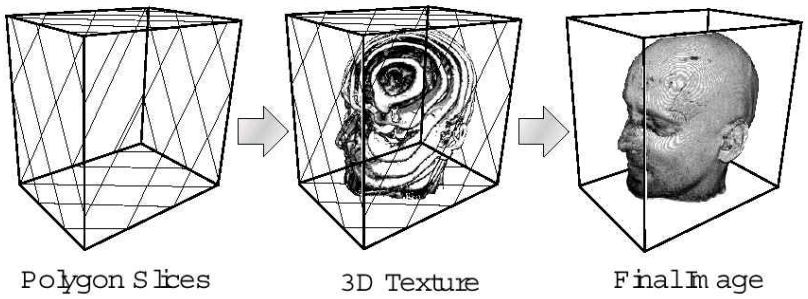
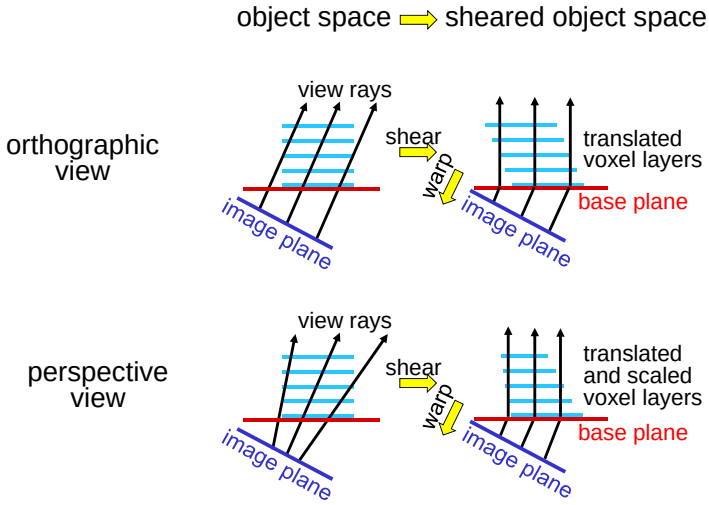


Image credit: H.W.Shen, Ohio State U.

4.2 Shear-Warp Factorisation

In general the image plane is not parallel to a volume face. The *shear-warp* method allows to render an intermediate image in the base plane:

1. Transform the *sheared object space* by translating (and possibly scaling) the voxel layers.
2. *Render* the intermediate image in the base plane.
3. *Warp* the intermediate image.



The *view transformation* is an affine transformation consisting of a rotation and a translation. Ignoring the translation, the 3×3 matrix can be factorised:

$$M_{\text{view}} = W \cdot S \cdot P$$

where

- P is a permutation matrix mapping the base plane to the xy plane.
- S is the shear matrix. The *shear* is of the form

$$S\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + z\begin{pmatrix} s_x \\ s_y \\ 0 \end{pmatrix}.$$

  With S being:

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

  where s_x and s_y have to be solved for from M_{view}.
- W is the warp matrix. The *warp* is a 3×3 matrix and effectively an affine transformation of the xy-plane. The third row of W is irrelevant while two zeros in the third column are required to make the warp independent of z.

$$W = \begin{pmatrix} w_{00} & w_{01} & 0 \\ w_{10} & w_{11} & 0 \\ w_{20} & w_{21} & w_{22} \end{pmatrix}$$

Assuming for simplicity that P is the identity, we get:

$$M_{\text{view}} = \begin{pmatrix} v_{00} & v_{01} & v_{02} \\ v_{10} & v_{11} & v_{12} \\ v_{20} & v_{21} & v_{22} \end{pmatrix} = W \cdot S = \begin{pmatrix} w_{00} & w_{01} & s_x w_{00} + s_y w_{01} \\ w_{10} & w_{11} & s_x w_{10} + s_y w_{11} \\ w_{20} & w_{21} & s_x w_{20} + s_y w_{21} + w_{22} \end{pmatrix}.$$

It follows for the warp coefficients $w_{ij} = v_{ij}$ $(j \neq 2)$ and for the shear coefficients:

$$\begin{pmatrix} s_x \\ s_y \end{pmatrix} = \begin{pmatrix} v_{00} & v_{01} \\ v_{10} & v_{11} \end{pmatrix}^{-1} \begin{pmatrix} v_{02} \\ v_{12} \end{pmatrix}$$

and for w_{22} (not needed):

$$w_{22} = -s_x v_{20} - s_y v_{21} + v_{22}.$$

If P is not the identity, permuted versions of S and W can be used.

4.3 Perspective shear warp

The same factorisation as before can be used. But now *homogeneous coordinates* are used:

$$M_{\text{view}} = W \cdot S \cdot P.$$

The *shear and scaling* matrix S gets the form

$$S = \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & s_w & 1 \end{pmatrix}.$$

It does

- a translation of x by $s_x z$ and of y by $s_y z$ followed by
- a scaling with $\frac{1}{1+s_w z}$.

The *warp* matrix now is

$$W = \begin{pmatrix} w_{00} & w_{01} & 0 & w_{03} \\ w_{10} & w_{11} & 0 & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & 0 & w_{33} \end{pmatrix}.$$

Assuming again that P is the identity we get:

$$M_{\text{view}} = \begin{pmatrix} v_{00} & v_{01} & v_{02} & v_{03} \\ v_{10} & v_{11} & v_{12} & v_{13} \\ v_{20} & v_{21} & v_{22} & v_{23} \\ v_{30} & v_{31} & v_{32} & v_{33} \end{pmatrix} = W \cdot S = \begin{pmatrix} w_{00} & w_{01} & s_x w_{00} + s_y w_{01} + s_w w_{03} & w_{03} \\ w_{10} & w_{11} & s_x w_{10} + s_y w_{11} + s_w w_{13} & w_{13} \\ w_{20} & w_{21} & s_x w_{20} + s_y w_{21} + w_{22} + + s_w w_{23} & w_{23} \\ w_{30} & w_{31} & s_x w_{30} + s_y w_{31} + s_w w_{33} & w_{33} \end{pmatrix}.$$

It follows that for the warp conditions $w_{ij} = v_{ij}$ with $(j \neq 2)$ holds. For the shear coefficients:

$$\begin{pmatrix} s_x \\ s_y \\ s_w \end{pmatrix} = \begin{pmatrix} v_{00} & v_{01} & v_{03} \\ v_{10} & v_{11} & v_{13} \\ v_{30} & v_{31} & v_{33} \end{pmatrix}^{-1} \begin{pmatrix} v_{02} \\ v_{12} \\ v_{32} \end{pmatrix}$$

and for w_{22} (not needed):

$$w_{22} = -s_x v_{20} - s_y v_{21} - s_w v_{23} + v_{22}.$$

For the shear-warp volume rendering algorithm now works as follows:

1. For each voxel layer (parallel to base plane):
   - Shear and scale the layer image by multiplying with S.
   - Apply the transfer functions.
2. Generate intermediate image with α compositing.
3. Warp the image by multiplying with W.

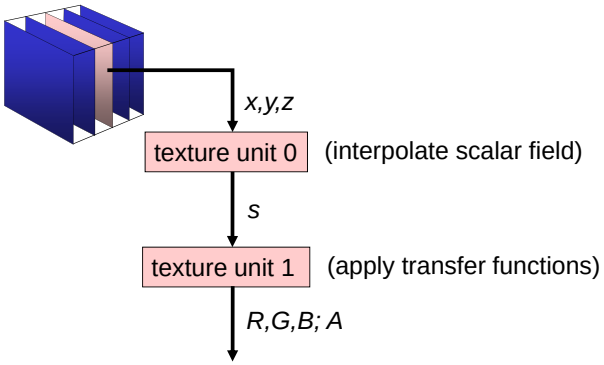
An advantage of this algorithm is that an aliasing filter can be used to prevent undersampling when scaling the image.

4.4 Object Space versus Image Space

Comparison of the object space methods introduced and image space methods such as *raycasting*.
Formally both methods are equivalent only the nesting order of the loops is different.
Practical differences:

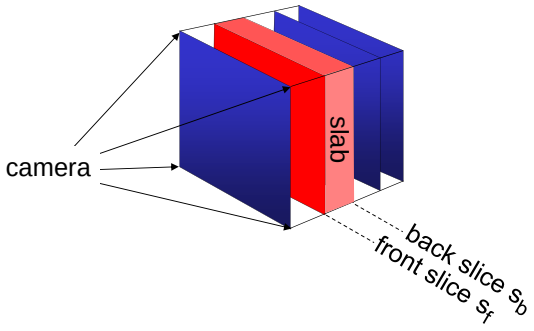
- Image space methods with FTB compositing allow early termination.
- Object space methods using a framebuffer for intermediate results suffer from quantisation artefacts.
- Object space methods can exploit texture mapping hardware and MIPmap textures for antialiasing.
- Image space methods would need supersampling in x and y to get antialiasing.

Post-classification The post-classification can be done directly in the graphics hardware. Using (OpenGL) *depndent texture* (two texture mapping stages):



Pre-integration It's also possible to pre-integrate in object space:

- Use *slabs* (space between two slices) and
- Dependent textures:
  - 1^{st} stage: Interpolate scalar filed in front and back slice.
  - 2^{nd} stage: Look up integrated transfer function.



4.5 Splatting

Raycasting "What does **each** voxel contribute to a given pixel?"

Splatting "What does **a given** voxel contribute to each pixel?"

Algorithm:

- Pre-processing:
  - For each voxel x_i render (raycast) a field $s(x_i) = \delta_{ij}$.
  - Store the resulting *footprint* images.
- Main loop:
  - For each voxel x_i adjust the footprint image to effective TF value.
  - Do α-compositing of all footprint images.

Advantages of splatting:

- Applicable to structured and unstructured grids.
- Other reconstruction filters than trilinear interpolation are possible (for example a sinc filter).

Original algorithm Westover 1990: Orthographic view and uniform grids. All footprints are translations of a template.

Sheet buffer method Westover 1991:

- Blend all footprint images of a voxel layer ("sheet buffer")
- Do α-compositing of sheet buffers.

Elliptical weighted average splatting (EWA), Zwicker et al. 2001:

- Ellipsoidal Gaussians are used as footprints
- Perspective view, low-pass filter for antialiasing.

4.6 Cell Projection

Projected tetrahedra (PT) is an object space method for tetrahedral grids [Shirley, Tuchman 1990]. Each (tetrahedral) cell is decomposed into 3 or 4 tetrahedra along those edges which are not part of the silhouette.

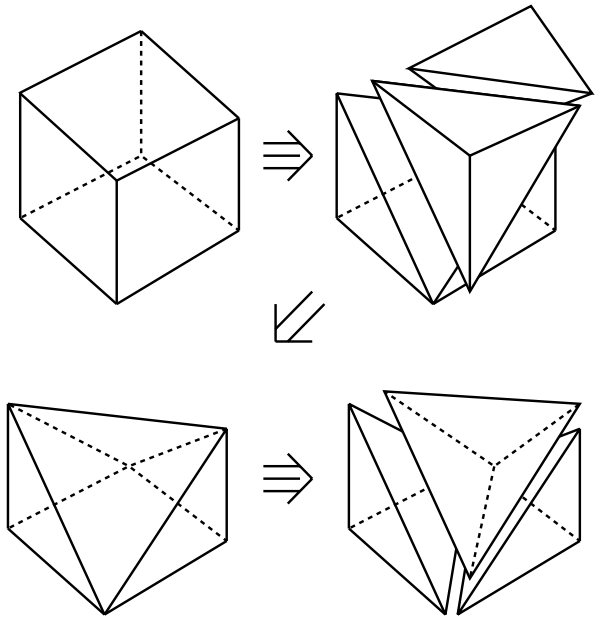


Figure 4: Decomposition of a cube into five tetrahedra. Source: A Polygonal Approximation to Direct Scalar Volume Rendering - [Shirley, Tuchmann 1990]

Cells are projected to *triangle fans* consisting of

- 1 *thick vertex* (projection of the common edge of the tetrahedra)
- 3 or 4 *thin vertices* (on the silhouette)

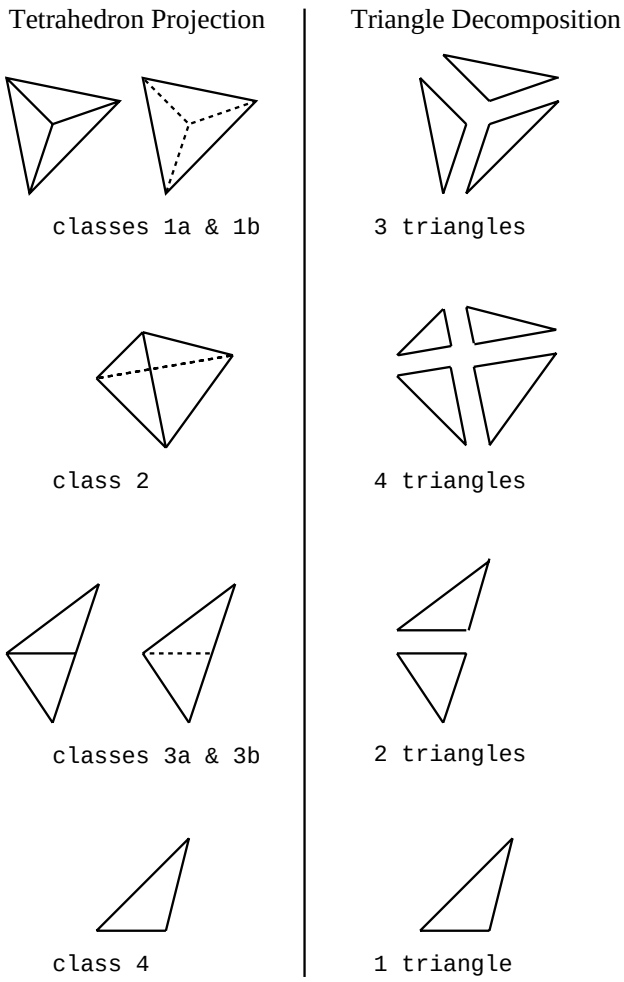


Figure 5: Splitting tetrahedra. Source: A Polygonal Approximation to Direct Scalar Volume Rendering - [Shirley, Tuchmann 1990]

Original algorithm: Store *triangle fan* in *space*:

- Thin vertices are kept in the *original position*.
- The thick vertex is set to the *midpoint* of the projected edge.

Advantages:

- *Depth test* can be used (allows volume rendering into a scene).
- *Viewing direction* and field-of-view can be changed (for a fixed camera position) by keeping the projection.

Computation of the thick vertex

- Compute the determinants

$$d_i = det(x_j, x_k, x_l) \qquad\qquad i \in [0, 3]$$

  where x_j, x_k and x_l are the vertices of the i^{th} face relative to camera position, ordered counter clockwise on the outside of the face.

- If the number of positive determinants is

  Odd Class 1

  Even Class 2

- Interpolate weights

Painting The thick and thin vertices are then sent to the graphics card (with supplied texture coordinates).

4.6.1 Drawing

Back-to-front compositing

- Cells must be depth-sorted ⇒ visibility sorting.
- Drawing possible without re-sorting on camera turn and zoom.
- Depth-test (z-buffer) must be enabled.
- Additional (opaque) objects must be rendered before the volumes.

Visibility sorting (MPVO algorithm, Williams 1992):

- Generate *partial ordering* of cells based on adjacent pairs.
- Break *cycles* (rare, small rendering error, alternative: Split a cell).
- Sort the list of *front cells* by distance to centroid (heuristic)

4.6.2 Example: Visualisation of smoke propagation

For this model we use a simple smoke model which is mostly used in fire protection engineering. The *absorption* τ is proportional to $s(x)$ (particle concentration). Which leads a to a simple preintegrated opacity transfer function:

$$\alpha = 1 - \exp\left(-c\frac{\tau_f + \tau_b}{2}\|x_b - x_f\|\right)$$

4.6.3 Opacities

When compositing cells with low opacity then opacities are essentially *added*. Adding many very small opacities (e.g. between $[0, 1/255]$) leads to *quantisation artefacts*. Options to reduce artefacts:

- Compositing with 16 bits
- *α-dithering* use *randomised rounding* instead of standard rounding:

$$x \to \lfloor x \rfloor + (x - \lfloor x \rfloor \geq \text{rand}) \qquad \text{with rand} \in [0, 1]$$

4.6.4 Hardware-Assisted Visibility Sorting

HAVS, Silva et al. 2005 is a faster cell projection algorithm:

- Requires 4 RGBA float buffers for storing 7 pairs of (s, d) per pixel:
  - With a scaler field value s,
  - and distance d to camera.
- The initial cell sorting is done on the CPU based on the centroids. Which results in a *k-nearly sorted sequence* with $k \leq 7$.
- Main loop: Draw all cell faces from back to front.
- Fragment shader:
  - Insert (s, d) into buffer
  - If the buffer is full:
    - Take out furthes pair of (s, d)
    - Compute thickness of cell behind the pixel:

$$\Delta d = d - d_{\text{old}}$$

    - Do a preintegrated TF lookup with s, s_{old}, Δd
    - Apply α compositing.

5 Vector Field Visualisation

- A *static vector field* $v(x)$ is a vector-valued function of space.
- A *time-dependent vector field* $v(x,t)$ also depends on time.

In the case of *velocity fields*, the terms *steady* and *unsteady flow* are used.
The dimensions of x and v are equal, often 2 or 3 and we denoted the components by $x,\ y,\ z$ and $u,\ v,\ w$:

$$x = (x,\ y,\ z), \qquad v = (u,\ v,\ w).$$

Sometimes a vector field is defined on a surface $x(i,j)$. The vector field is then a function of parameters and time:

$$v(i,j,t).$$

5.1 Visualisation

An elementary visualisation of a vector field is to *draw arrows*:

- At the data points (gird nodes or cell centers), or
- at a new (uniform) grids. For 3D fields it's often a 2D slice.

Arrows can visualise:

- Direction
- Relative magnitude (when appropriately scaled(
- Time dependency (when animated)

Problems

- It is not clear whether arrows represent vector values at the start point or at the midpoint of the arrow.
- Of there exist no satisfactory scaling factors:
  - Large scaling: Arrows occlude each other
  - Small scaling: Direction is not recognisable in some regions
  - Fixed length: Magnitued information is lost.

5.2 Vector fields as ODEs

For simplicity, the vector field is now interpreted as *velocity field*. The field $v(x,t)$ describes the connection between location and velocity of a (massless) particle. Which can equivalently be expressed as an *ordinary differential equation*:

$$\dot{x}(t) = v(x(t), t).$$

This ODE, together with an *initial condition*

$$x(t_0) = x_0,$$

is a so-called *initial value problem* (IVP). Its solution is the *integral curve* (or *trajectory*)

$$x(t) = x_0 + \int_{t_0}^{t} v(x(\tau), \tau) d\tau.$$

The integral curve is a *pathline* decribing the *path* of a massless *particle* which was released at time t_0 at position x_0.
Remark: $t < t_0$ is allowed.
For static fields the ODE is *autonomous*:

$$\dot{x}(t) = v(x(t))$$

and its integral curves

$$x(t) = x_0 + \int_{t_0}^{t} v(x(\tau)) d\tau$$

are called *field lines* or in the case of velocity fields *streamlines*.

- In *static vector fields* pathlines and streamlines are *identical*.
- In *time-dependent* vector fields *instantaneous streamlines* can be computed from a "snapshot" at a fixed time T (which is a static vector field):

$$v_T(x) = v(x, T).$$

In practice time-dependent fields are often given as a dataset per time step. Each dataset is then a snapshot.
Computing streaklines or timelines is more expensive than solving a single IVP.

5.2.1 Pathlines

Pathlines are the trajectories that individual fluid particles follow. These can be thought of as "recording" the path of a fluid element in the flow over a certain period. The direction the path takes will be determined by the streamlines of the fluid at each moment in time.[2]

[2] https://en.wikipedia.org/wiki/Streamlines,_streaklines,_and_pathlines

- Are physically meaningful
- Allow comparison with experiment (observe marked particles)
- Are well suited for dynamic visualisation (of particles).

5.2.2 Streamlines

Streamlines are a family of curves that are instantaneously tangent to the velocity vector of the flow. These show the direction a fluid element will travel in at any point in time.[3]

- Are only geometrically but not physically meaningful,
- Are easiest to compute (no temporal interpolation, single IVP),
- Are better suited for static visualisation (prints),
- Don't intersect (under reasonable assumptions)

Inputs

- Static vector field $v(x)$
- Seed points with time of release (x_0, t_0)
- Control Parameters:
  - Step size (temporal, spatial or in local coordinates)
  - Step count limit, time limit, etc...
  - Order of integration scheme

Output Streamlines as "polylines", with possible attributes.

Preprocessing

- Set up search structure for point location
- For each seed point:
  - *Global point location*:
    Given a point x, find the cell containing x and the local coordinates (ξ, ν, ζ).
    Alternatively if the grid is structured find the computational space coordinates

    $$(i + \xi, j + \nu, k + \zeta)$$

  - If x is not found in a cell then remove the seed point.

[3] https://en.wikipedia.org/wiki/Streamlines,_streaklines,_and_pathlines

Integration For each seed point x:

- Interpolate v trilinearly to local coordinates (ξ, ν, ζ)
- Do an integration step producing a new point x'
- *Incremental point location*: For position x' find cell and local coordinates (ξ', ν', ζ') making use of information (coordinates, local coordinates, cell) of old point x.

Integration step Widely used methods:

- *Euler*, inaccurate, used only in special speed-optimised techniques:

$$x_{\text{new}} = x + v(x, t) \cdot \Delta t$$

- *Runga-Kutta*, 2^{nd} or 4^{th} order. Higher order schemes are often too slow for visualisation. Yeung/Pope 1987 showed that when using standard trilinear interpolation *interpolation errors* dominate *integration errors*.
- There are several options for choosing a step size:
  - Fixed time stepping Δt
  - Fixed spatial step Δs:

    The time step is derive from he spatial step

$$\Delta t = \frac{\Delta s}{\|v(x)\|}$$

    which needs to be corrected iteratively and is used for methods such as LIC.
  - Adaptive:
    - Adapting to grid resolution (such as cell size)
    - Adapting to data variation (Runge-Kutta-Fehlberg method)
    - Used for interactive viewing (with zooming)

Termination criteria

- Grid boundary reached
- Step count limit reached
- Optional: Velocity close to zero
- Optional: Time limit reached
- Optional: Arc length limit reached

5.2.3 Streaklines

Streaklines are the locus of points of all the fluid particles that have passed continuously through a particular spatial point in the past. Dye steadily injected into the fluid at a fixed point extends along a streakline.

- Are physically meaningful
- Allow a direct comparison with the experiment (i.e. dye injection)
- Are well suited for static and dynamic visualisation
- Are a good choice for fast moing vortices
- Can be approximated by a set of disconnected particles.

Algorithm:

1. For time samples t_0, t_1, ..., t_n solve the IVP
$$\begin{aligned} \dot{x}_i(t) &= v(x_i(t), t) \\ x_i(t_i) &= y \end{aligned}$$
2. Extract the point $x_i(t_n)$)

In numerical computation the temporal interval must be *adaptively refined* if two successive particles diverge too much.

5.2.4 Timelines

Timelines are the lines formed by a set of fluid particles that were marked at a previous instant in time, creating a line or a curve that is displaced in time as the particles move.

- Are physically meaningful
- Are well suited for static and dynamic visualisation
- Can be approximated by a set of disconnected particles.

Algorithm:

1. For point samples y_0, y_1, ..., y_n on the seed curve solve the IVP:
$$\begin{aligned} \dot{x}_i(t) &= v(x_i(t), t) \\ x_i(t_0) &= y_i \end{aligned}$$
2. From the integral curve $x_i(t)$ extract the point $x_i(T)$.
3. Connect these points.

The result is a timeline for time T.
In the numerical computation the spatial interval must be *adaptively refined* if two neighbour particles diverge too much.

5.3 The Stencil Walk algorithm

Computing the *incremental point location* is nontrivial for curvilinear and unstructured grids.
Buning's *stencil walk* algorithm solves this provlem.

Given

- Point with coordinates x
- Cell (as three parameters (i, j, k) or as index c respectively)
- Local coordinates (ξ, ν, ζ)
- Coordinates of a new point x'

Wanted

- New cell as (i', j', k') or as index c' respectively
- New local coordinates (ξ, ν, ζ)

Algorithm In a first phase the algorithm find the cell containing x' by iteratively doing:

- Take the difference vector $\Delta x = x' - x$
- Intersecting the ray $x + t\Delta x$ with the cell boundary giving a t value.
  - Linearise the coordinate transform

    $$\varphi : (\xi, \nu, \zeta) \mapsto (x, y, z)$$

    in the point $x = (x, y, z)$ by computing the *Jacobian*

    $$J = \frac{\delta(x, y, z)}{\delta(\xi, \nu, \zeta)} = \left[\frac{\delta\varphi}{\delta\xi} | \frac{\delta\varphi}{\delta\nu} | \frac{\delta\varphi}{\delta\zeta}\right].$$

  - Using the *Jacobian* to transform the difference vector $\Delta x = x' - x$ into the local coordinate frame of the cell:

    $$(\Delta\xi, \Delta\nu, \Delta\zeta) = J^{-1}\Delta x.$$

  - Find the intersection of the ray

    $$(\xi, \nu, \zeta) + t(\Delta\xi, \Delta\nu, \Delta\zeta)$$

    with the cell boundary having the equations:

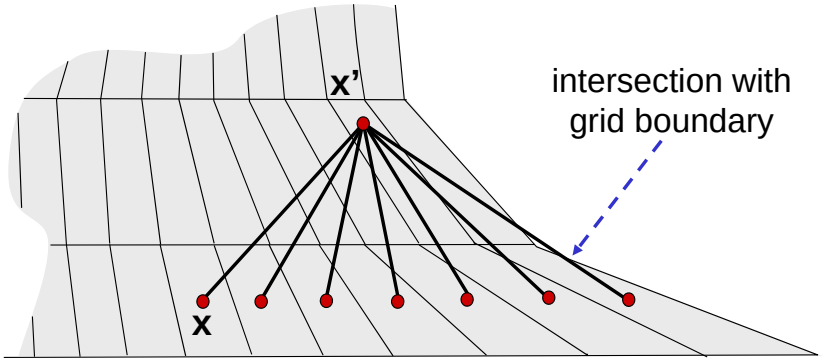
$$\xi, \nu, \zeta = 0 \tag{1}$$
$$\xi, \nu, \zeta = 1 \quad \text{for hex cell} \tag{2}$$
$$\xi + \nu + \zeta = 1 \quad \text{for tet cell} \tag{3}$$
$$0 \leq \xi, \nu, \zeta \geq 1 \quad \text{inequalities} \tag{4}$$

Due to linearisation the point is not exact and in most cases the correct neighbour cell is found.

- If $t \geq 1$ the point x' lies in the current cell and iteration can be stopped.
- Otherwise move to the neighbour cell adjacent at the intersection point
- If no such cell exists terminate with failure
- Set the *cell centroid* as the new x for the next iteration.

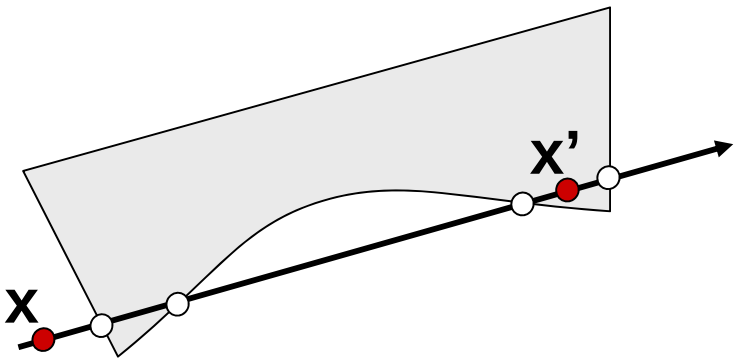
5.3.1 Problems

If the cells are sufficiently skewed the algorithm can walk away from the target cell.



This problem can be solved with a *modification* of the algorithm:

- Keep ray (x, x') unchanged
- New Problem: Cell faces of the type *quadrangle* (nonplanar!) can be intersected *twice*!



Therefore *exact* intersection points of the ray with bilinear surface patches must be calculated:

Exact intersection calculation If the quadrangle is *nonplanar* the four corners P_0, P_1, P_2, P_3 can be mapped to:

$$\tilde{P}_0 = (0,0,0)$$
$$\tilde{P}_1 = (1,0,0)$$
$$\tilde{P}_2 = (1,1,1)$$
$$\tilde{P}_3 = (0,1,0)$$

by the *affine transformation*

$$\tilde{x} = (\tilde{P}_1|\tilde{P}_2|\tilde{P}_3)(P_1|P_2|P_3)^{-1}(x - P_0).$$

The bilinear surface containing P'_0, P'_1, P'_2 and P'_3 is the *hyperbolic paraboloid*

$$z = xy.$$

Inserting the transformed view ray $\tilde{x} + t\Delta\tilde{x}$ leads to a quadratic equation for t:

$$(\tilde{z} + t\Delta\tilde{z}) = (\tilde{x} + t\Delta\tilde{x})(\tilde{y} + t\Delta\tilde{y}).$$

If real solutions with $0 < t < 1$ exist the intersection points $(\tilde{x}_i, \tilde{y}_i, \tilde{z}_i)$ are computed and transformed back.
In the second phase the stencil walk algorithm computes the local coordinates of the point in the cell known to contain it. The local coordinates are the inverse of the coordinate function:

$$\varphi : (\xi, \nu, \zeta) \mapsto (x, y, z)$$

evaluate at the given point

$$x = (x, y, z).$$

However, the trilinear function π is a cubic polynomial and its inverse is a sixth-degree polynomial. Hence the problem needs to be solved with Newton's method.

5.4 Global Point Location

Global point location is more expensive than the stencil walk algorithm.
Many methods trade in safety for efficiency. A few methods are:

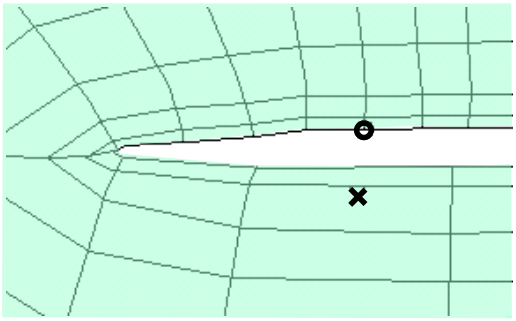
1. Search for the point in every grid cell using Newton's Method.

   Hypothetic "brute force" method. Safe.

2. Buning's method.

   Do incremental point location starting from a boundary cell. **Problem**:

Node (O) nearest to given point (X) is not necessarily adjacent to the cell containing it. Furthermore the straight line between the two points can leave the grid.



Bunings method is safe if incremental search is repeated with a different boundary cell as long as the point is not found. Instead of using all boundary cells, precompute a subset of boundary cells to guarantee to find all points within the grid.

3. Incremental point location starting from a node near the grid center.

   Simple method. Only safe for *star-shaped* grids.

4. Search structure based.

   Efficient methods use a search structure such as a uniform grid, octrees or kd-trees for nodes or cell centers:

   - When a point query is not sufficient a *range query* is need with a range determined by the cell size.
   - Problem: Cells can have extreme aspect ratios (especially from CFD).

5. Bounding box hierarchy

   Use a bounding box hierarchy for a recursively subdivided grid. Efficient and safe method and easy for structured grids.

   More preprocessing is required for unstructured grids (cell tree, Garth 2010)

5.5 Computational Space Streamline Integration

In structured grids *point location* can be *avoided* by using a different approach:
Integration can be done in computational space $\mathcal{C}$ instead of physical space $\mathcal{P}$. This requires a modification of the integration algorithm:

- *Before* the integration step:

  Transform the *velocity* $v(x)$ to $\mathcal{C}$ by multiplying with J^{-1}

- *After* the integration step:

  This step is only required if graphical output of this step is needed. Transform the new *position*

  $$x = (i + \xi, j + \nu, k + \zeta)$$

  to $\mathcal{P}$ by trilinear interpolation.

Main problem of the integration in $\mathcal{C}$:

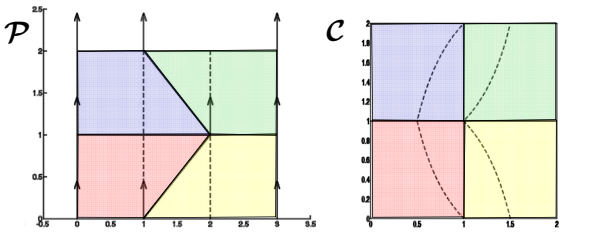
- The ordinate function

  $$\varphi : (i + \xi, j + \nu, k + \zeta) \mapsto (x, y, z)$$

  is only C^0 continuous at cell boundaries.

- Therefore J is discontinous.

- Example (Sadarjoen 1994): Four cells with a constant velocity field.



straight streamlines (dashed) jagged streamlines (dashed)

Two sources of error:

- Integration steps across cell boundaries.

  This can be avoided by shortening such steps.

- Use of a (precomputed) single transformed field vector by node.

  Can be fixed by transforming all eight vectors of a cell on the fly when entering a new cell.

The main advantage of integration in $\mathcal{C}$ is its algorithmic simplicity. But if it's done correctly (avoiding the errors above) it can be slower than an integration in $\mathcal{P}$.

5.6 Skin Friction Lines

Velocity fields of fluid flow can have grid boundaries which are walls (i.e. solid material surfaces). At walls the velocity vector is usually zero as a result of the no-slip boundary condition. Therefore a derived vector field is often used: The *wall shear stress*:

$$\tau_w = \mu \frac{\delta v_w}{\delta s},$$

can be obtained as the limit of the wall-parallel velocity component v_w divided by the wall distances s and multiplied with the *dynamic viscosity* μ (material constant). This limit is typically nonzero except at isolated points.
Streamlines of τ_w are called *skin friction lines*. They are an example of a vector field defined on a surface in 3-space.

5.7 Streamline Placement

The problems of visualisation by streamlines:

- Dependency on seed points,
- and the density of streamlines can be largely inhomogogeneous.

Solution Automatically optimise choice of seed points.

1. *Streamlets* (short streamline segments)

   The length is proportional to the velocity magnitude (obtained automatically by using a fixed integration time).

   Start with a uniform grid and make spacing roughy even by locally adapting (displacing, inserting, removing) seeds.

   Typical use: Displaying weather maps

2. Algorithm by Turk and Bank (longer streamlines):

   Objective: Create a streamline image which when low-pass filtered has a uniform grey level.

   Optimise seed positions and integration lengths.

   Operations:

   - Insert
   - Delete
   - Move
   - Lengthen
   - Shorten

   Apply operations either randomly or based on oracles.

5.8 Streamsurfaces

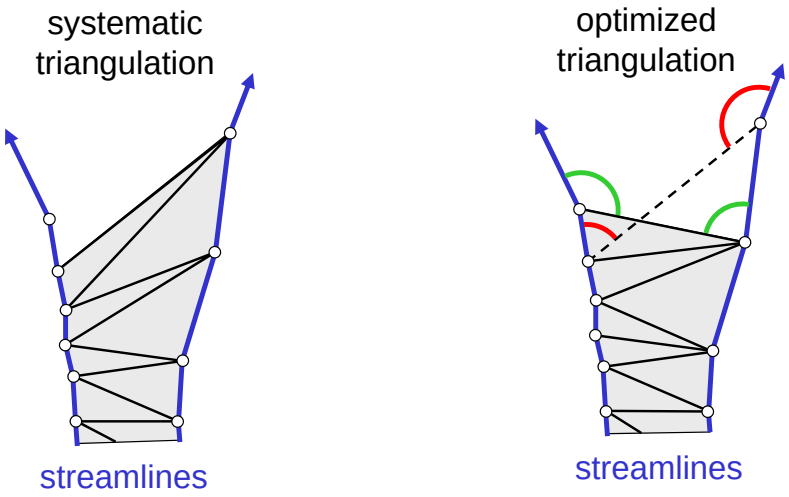
Definition: Union of streamlines seeded densly on a curve.
Advantage for visualisation: Structured, better spatial perception.
Naive algorithm:

- Start integration at discrete samples on the seed curve.

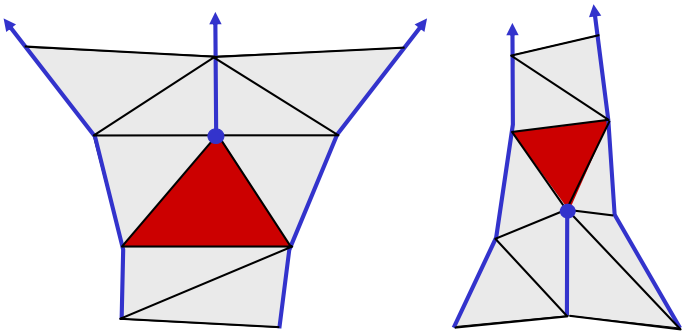
- Conncect points of equal integration time resulting in a quad mesh.

This fails if streamlines diverge or grow at largely different speeds.

Hultquist's algorithm solves this problem of speed differences by *optimised triangulation*: Of two possible connections choose the one which is closer to orthogonal (sic) to both streamlines.



The problem of divergence or convergence is solved by *inserting* or *terminating streamlines*.



inserted streamline terminated streamline

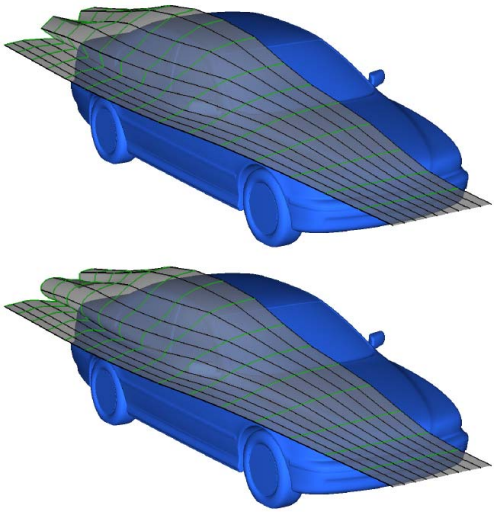
Also use hermite interpolation for streamsurface patches:

linear interpolation

vs.

Hermite-interpolation

of front curves.
(Schneider et al., 2009)



5.9 Streak Surfaces

Stream lines in 3D on a fully adaptive triangle mesh.