CS 247 – Scientific Visualization
Lecture 9: Scalar Fields, Pt. 5 [preview]

Markus Hadwiger, KAUST

Reading Assignment #5 (until Feb 28)

Read (required):

- Gradients of scalar-valued functions

  `https://en.wikipedia.org/wiki/Gradient`

- Critical points

  `https://en.wikipedia.org/wiki/Critical_point_(mathematics)`

- Multivariable derivatives and differentials

  `https://en.wikipedia.org/wiki/Total_derivative`

  `https://en.wikipedia.org/wiki/Differential_of_a_function#Differentials_in_several_variables`

  `https://en.wikipedia.org/wiki/Hessian_matrix`

- Dot product, inner product (more general)

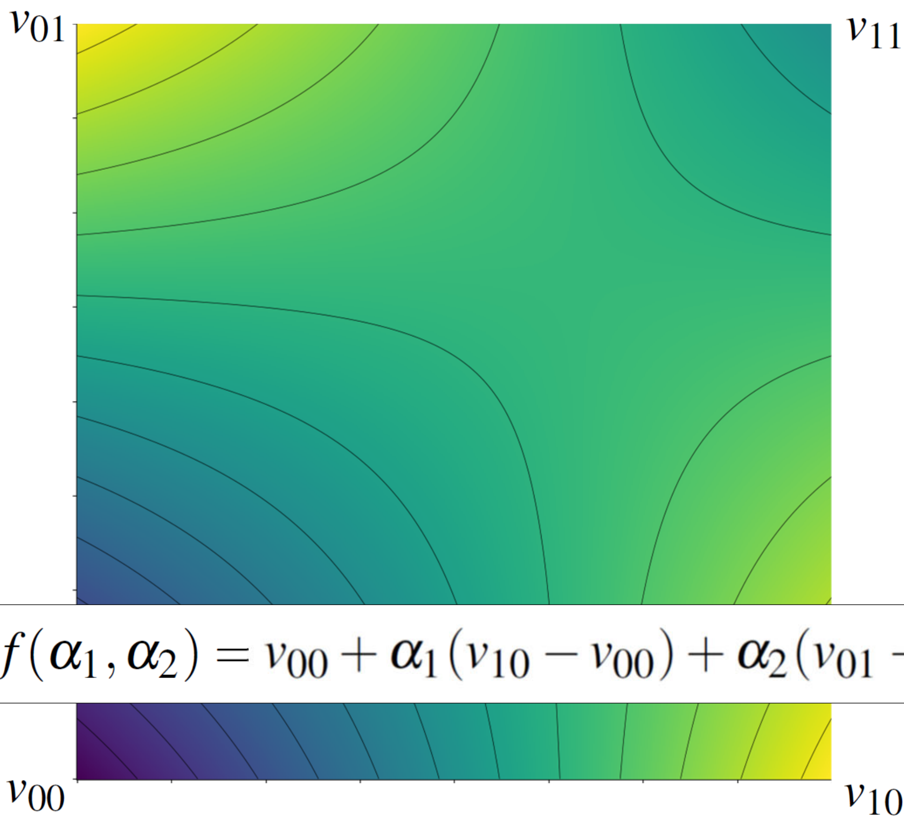
  `https://en.wikipedia.org/wiki/Dot_product`

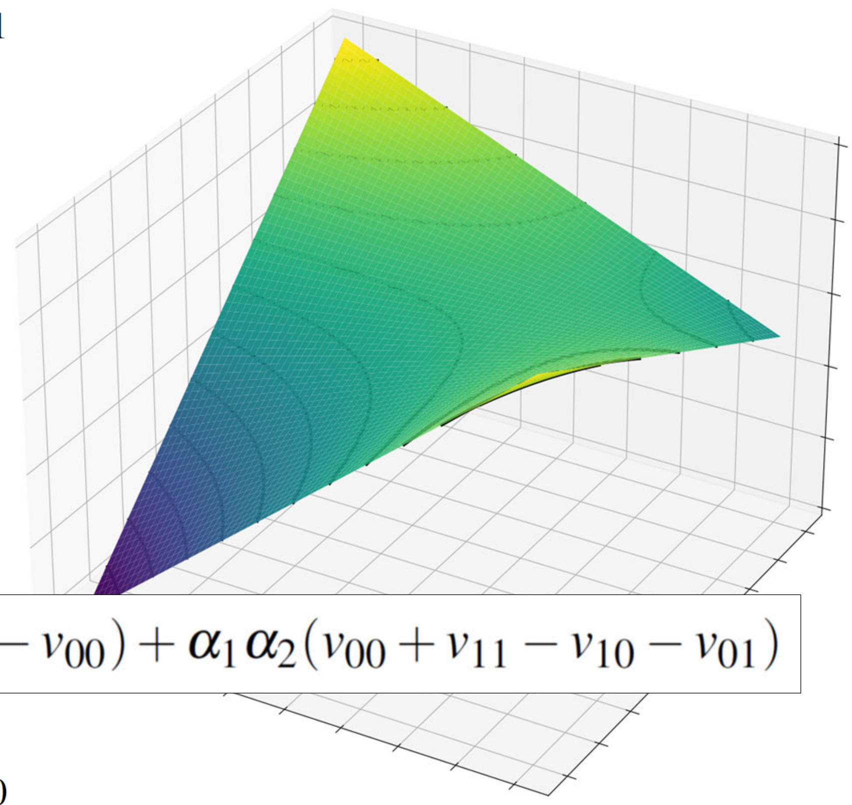
  `https://en.wikipedia.org/wiki/Inner_product_space`

Bi-Linear Interpolation

Consider area between 2x2 adjacent samples

Example: 1.0 at top-left and bottom-right, 0.0 at bottom-left, 0.5 at top-right

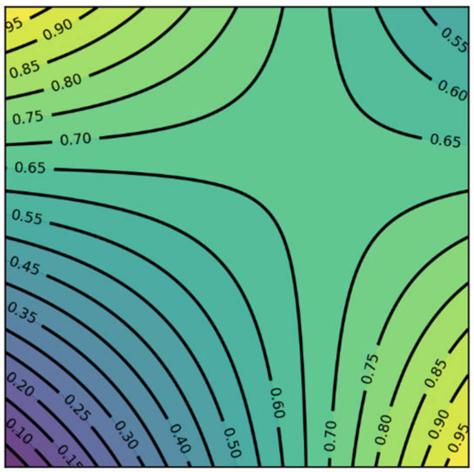




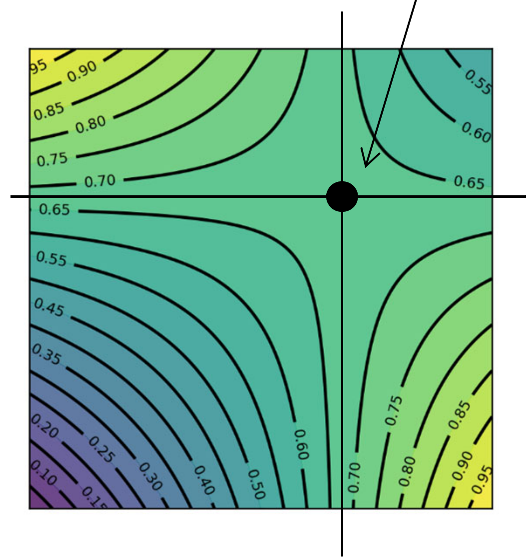
$$f(\alpha_1, \alpha_2) = v_{00} + \alpha_1 (v_{10} - v_{00}) + \alpha_2 (v_{01} - v_{00}) + \alpha_1 \alpha_2 (v_{00} + v_{11} - v_{10} - v_{01})$$

Bi-Linear Interpolation: Critical Points

Critical points are where the gradient vanishes (i.e., is the zero vector)



critical point
(saddle point)



here, the critical
value is 2/3=0.666…

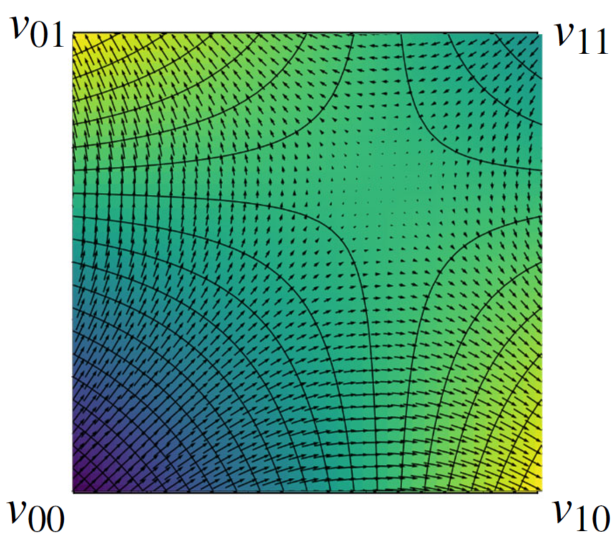
“Asymptotic decider”: resolve ambiguous configurations (6 and 9) by
comparing specific iso-value with critical value (scalar value at critical point)

Preview: Critical Point and Value (Details Later)

Compute gradient (critical points are where gradient is zero vector):

$$\frac{\partial f(\alpha_1,\alpha_2)}{\partial \alpha_1} = (v_{10} - v_{00}) + \alpha_2(v_{00} + v_{11} - v_{10} - v_{01})$$

$$\frac{\partial f(\alpha_1,\alpha_2)}{\partial \alpha_2} = (v_{01} - v_{00}) + \alpha_1(v_{00} + v_{11} - v_{10} - v_{01})$$

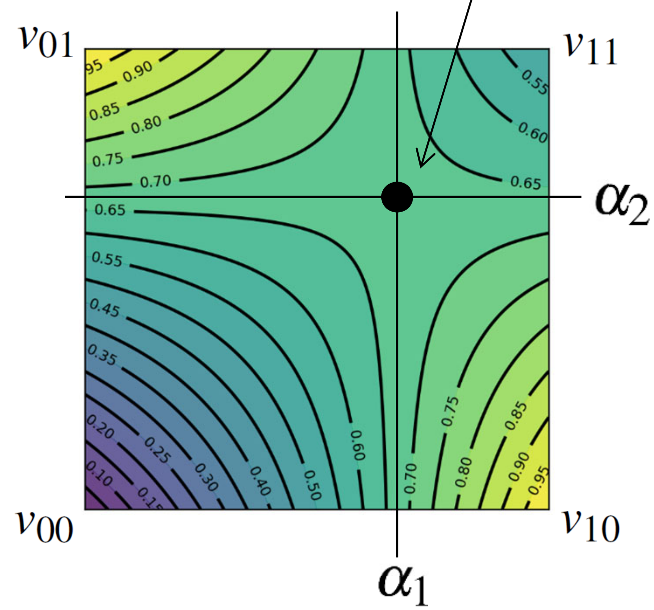


Preview: Critical Point and Value (Details Later)

Where is the critical point, and what is the critical value?

$$\frac{\partial f(\alpha_1,\alpha_2)}{\partial \alpha_1} = 0: \qquad \alpha_2 = \frac{v_{00} - v_{10}}{v_{00} + v_{11} - v_{10} - v_{01}}$$

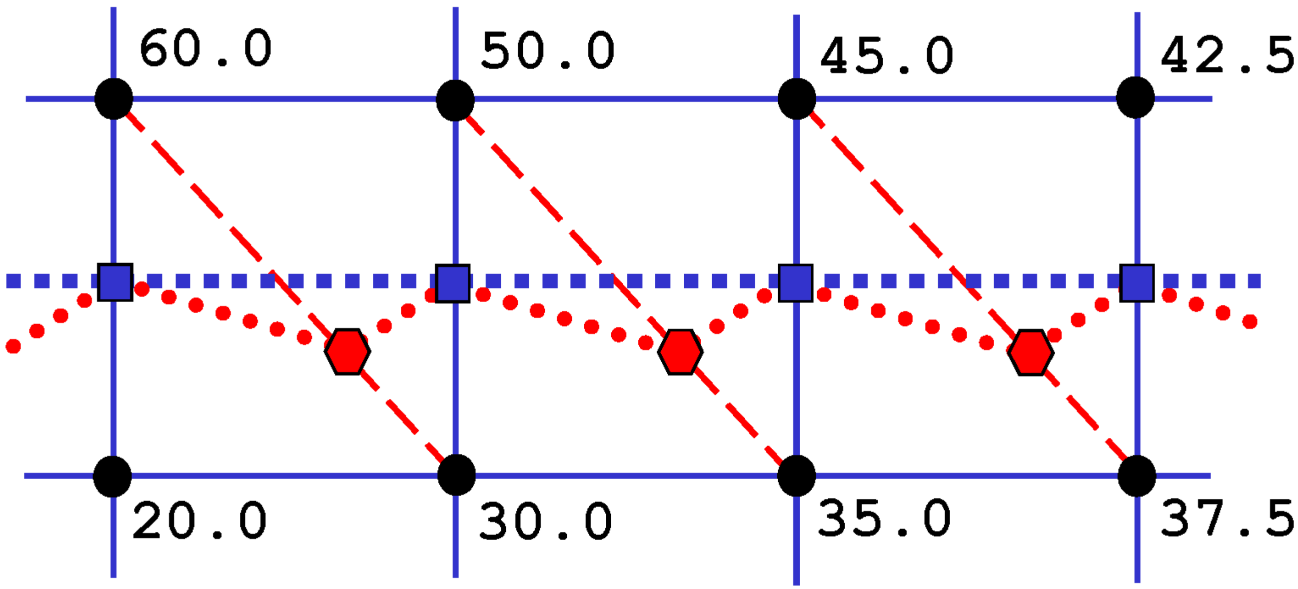
$$\frac{\partial f(\alpha_1,\alpha_2)}{\partial \alpha_2} = 0: \qquad \alpha_1 = \frac{v_{00} - v_{01}}{v_{00} + v_{11} - v_{10} - v_{01}}$$



$$f(\alpha_1,\alpha_2) = v_{00} + \alpha_1(v_{10} - v_{00}) + \alpha_2(v_{01} - v_{00}) + \alpha_1\alpha_2(v_{00} + v_{11} - v_{10} - v_{01})$$

Contours in triangle/tetrahedral cells

Illustrative example: Find contour at level *c*=40.0 !



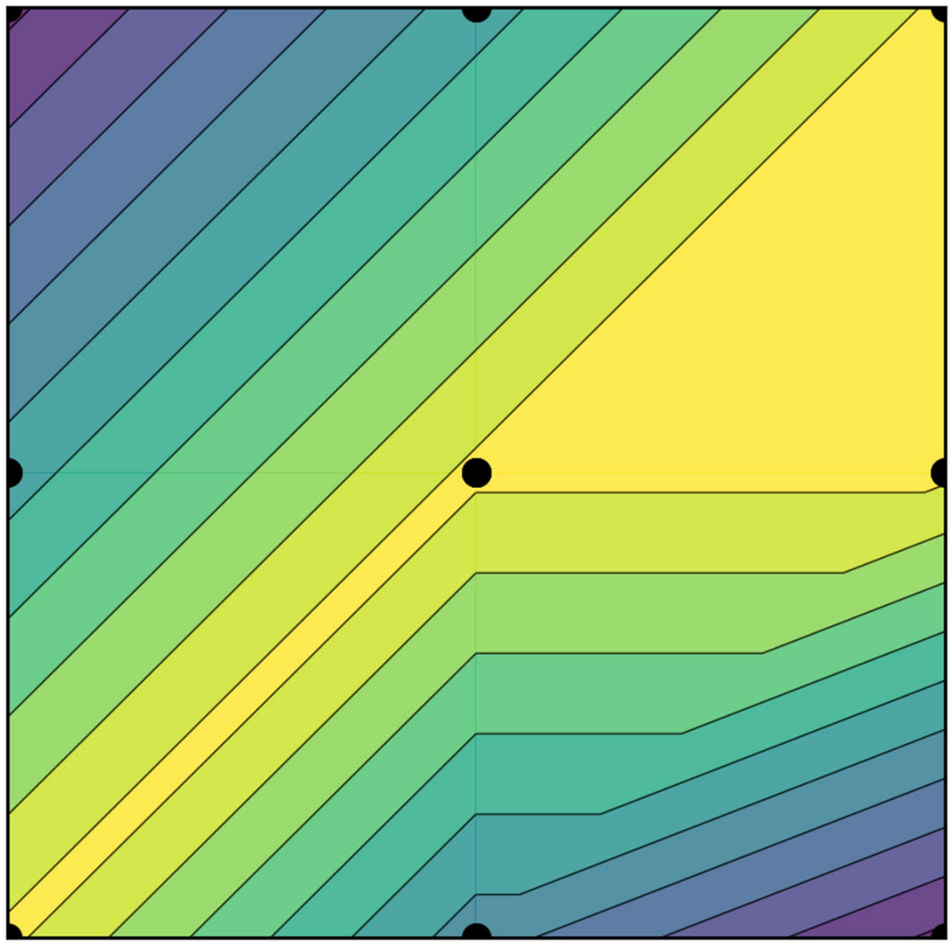
original quad grid, yielding vertices and contour

triangulated grid, yielding vertices and contour

Bi-Linear Interpolation: Comparisons

linear

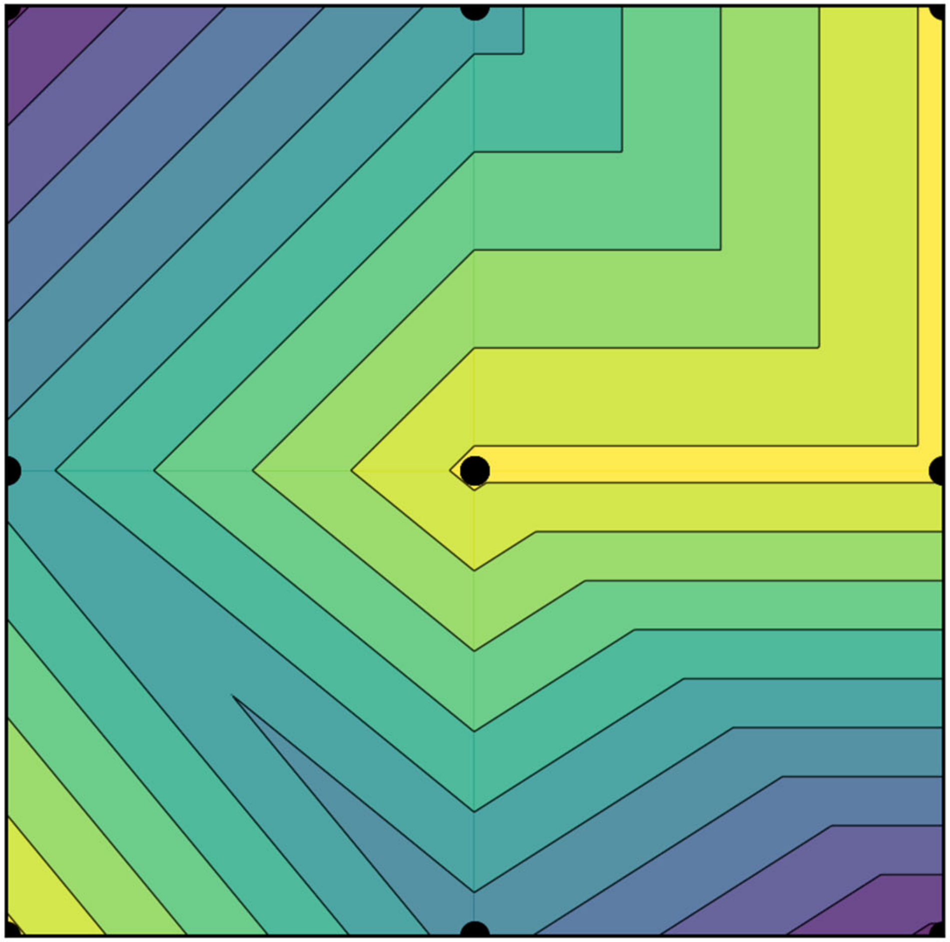
(2 triangles per quad;
diagonal:
bottom-left,
top-right)



Bi-Linear Interpolation: Comparisons

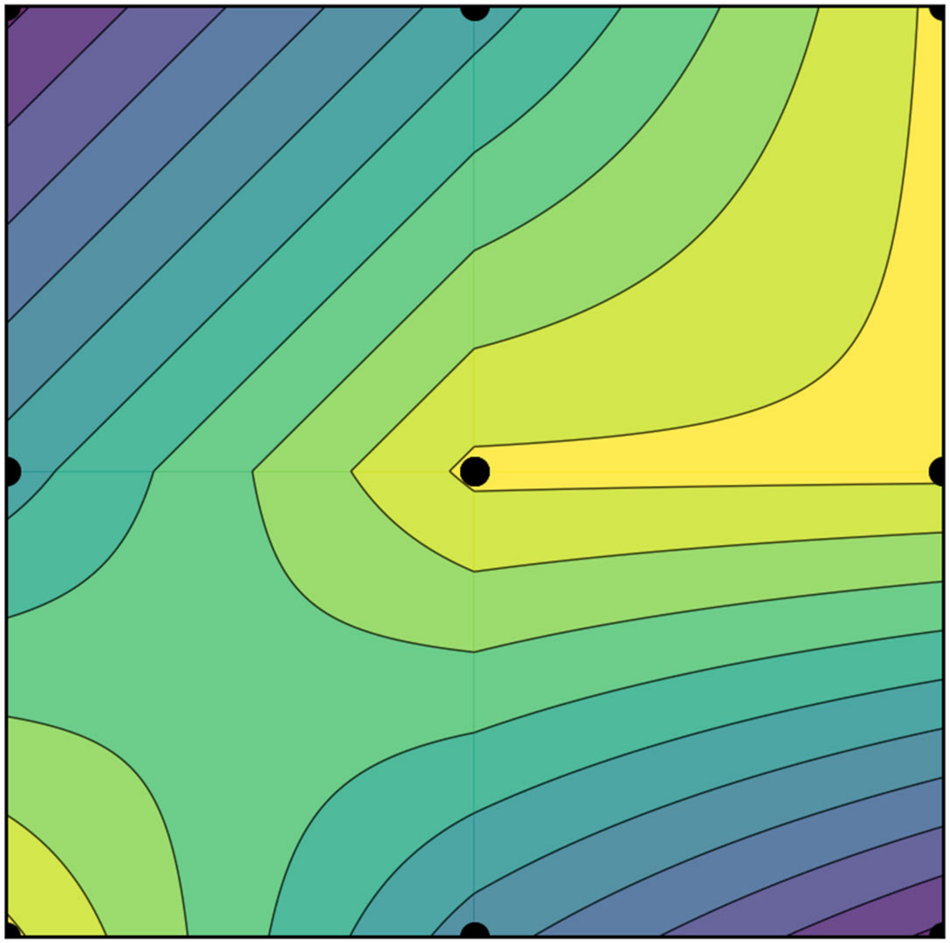
linear

(2 triangles per quad;
diagonal:
top-left,
bottom-right)



Bi-Linear Interpolation: Comparisons

bi-linear



From 2D to 3D (Domain)

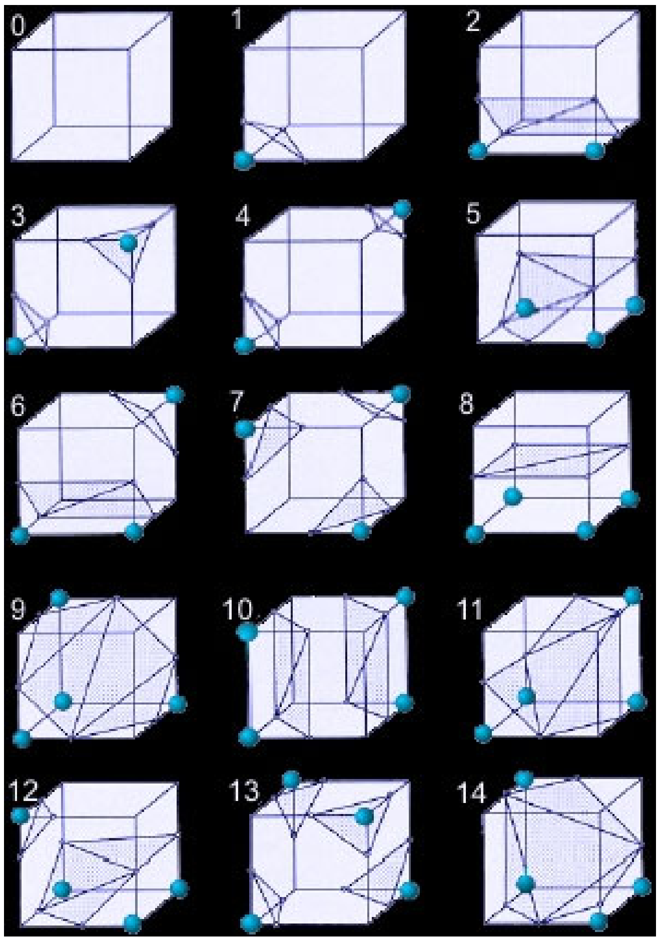
2D - Marching Squares Algorithm:

1. Locate the contour corresponding to a user-specified iso value
2. Create lines

3D - Marching Cubes Algorithm:

1. Locate the surface corresponding to a user-specified iso value
2. Create triangles
3. Calculate normals to the surface at each vertex
4. Draw shaded triangles

Marching Cubes



- For each cell, we have 8 vertices with 2 possible states each (inside or outside).
- This gives us 2^8 possible patterns = 256 cases.
- Enumerate cases to create a LUT
- Use symmetries to reduce problem from 256 to 15 cases.

Explanations

- Data Visualization book, 5.3.2
- Marching Cubes: A high resolution 3D surface construction algorithm, Lorensen & Cline, ACM SIGGRAPH 1987

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, a 8-bit number is computed from the 8 signs of $\tilde{f}(x_i)$ on the corners of a hexahedral cell.
- The isosurface piece is looked up in a table with 256 entries.

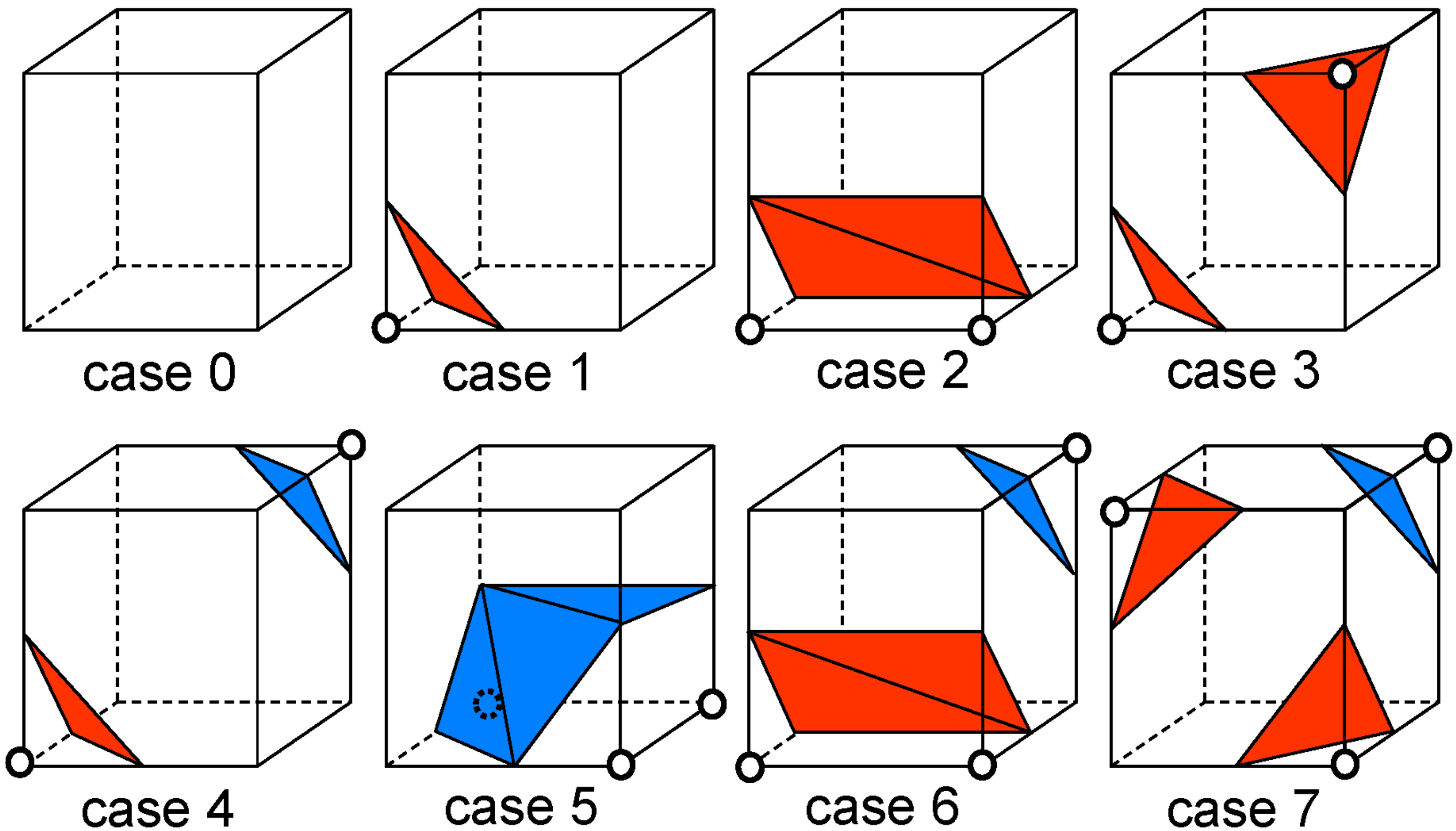
How to build up the table of 256 cases?

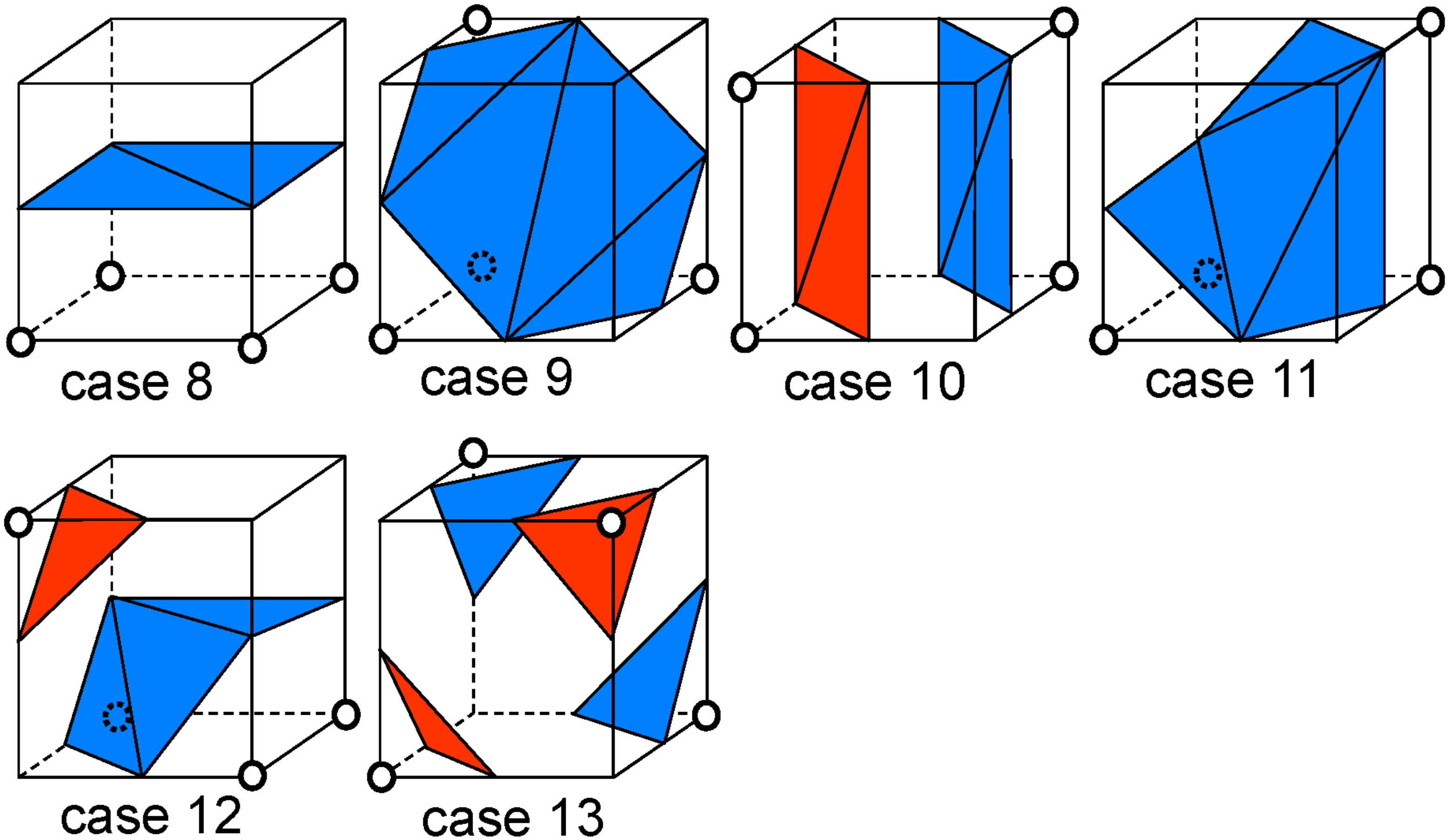
Lorensen and Cline (1987) exploited 3 types of symmetries:
- rotational symmetries of the cube
- reflective symmetries of the cube
- sign changes of $\tilde{f}(x_i)$

They published a reduced set of 14*) cases shown on the next slides where
- white circles indicate positive signs of $\tilde{f}(x_i)$
- the positive side of the isosurface is drawn in red, the negative side in blue.

*) plus an unnecessary "case 14" which is a symmetric image of case 11.

The marching cubes algorithm




case 8
case 9
case 10
case 11
case 12
case 13

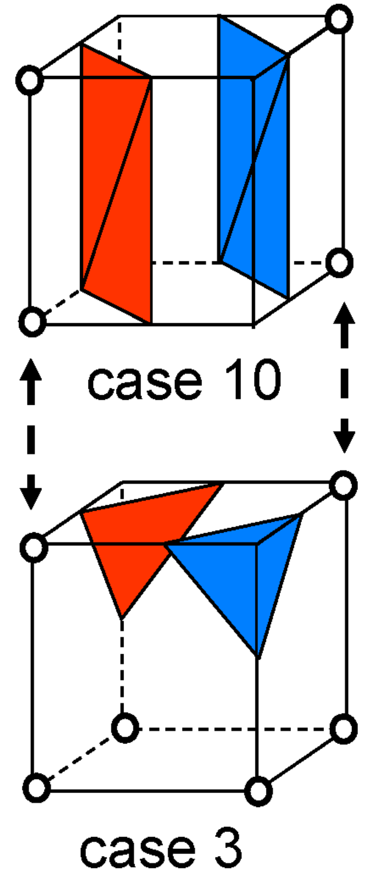
Do the pieces fit together?

- The correct isosurfaces of the trilinear interpolant would fit (trilinear reduces to bilinear on the cell interfaces)
- but the marching cubes polygons don't necessarily fit.

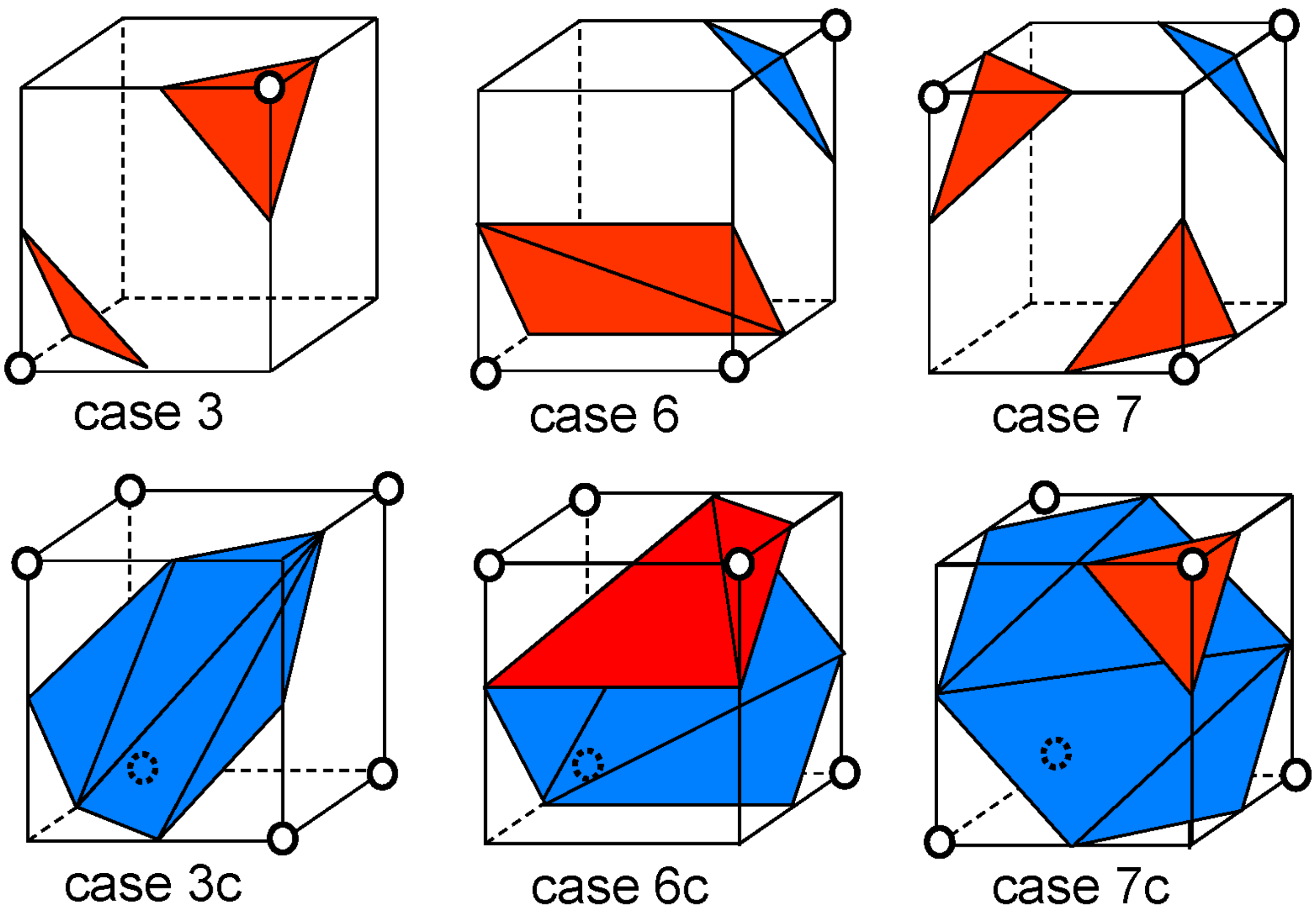
Example

- case 10, on top of
- case 3 (rotated, signs changed)

have matching signs at nodes but polygons don't fit.



The marching cubes algorithm



Summary of marching cubes algorithm:

Pre-processing steps:

- build a table of the 28 cases
- derive a table of the 256 cases, containing info on
  - intersected cell edges, e.g. for case 3/256 (see case 2/28): (0,2), (0,4), (1,3), (1,5)
  - triangles based on these points, e.g. for case 3/256: (0,2,1), (1,3,2).

Loop over cells:

- find sign of $\tilde{f}(x_i)$ for the 8 corner nodes, giving 8-bit integer
- use as index into (256 case) table
- find intersection points on edges listed in table, using linear interpolation
- generate triangles according to table

Post-processing steps:

- connect triangles (share vertices)
- compute normal vectors
  - by averaging triangle normals (problem: thin triangles!)
  - by estimating the gradient of the field $f(x_i)$ (better)

Triangle Mesh Data Structure (1)

Store list of vertices; vertices shared by triangles are replicated

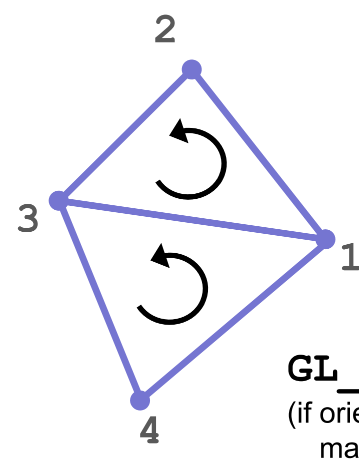
Render, e.g., with OpenGL immediate mode, …

coords for vertex 1

```
x1,y1,z1  ┐
x2,y2,z2  │ face 1
x3,y3,z3  ┘
x1,y1,z1  ┐
x3,y3,z3  │ face 2
x4,y4,z4  ┘

...
```

```
struct face
  float verts[3][3]
  DataType val;
```



GL_CCW
(if orientable manifold)

Redundant, large storage size, cannot modify shared vertices easily

Store data values per face, or separately

Triangle Mesh Data Structure (2)

Indexed face set: store list of vertices; store triangles as indexes

Render using separate vertex and index arrays / buffers

coords for vertex 1

vertex list

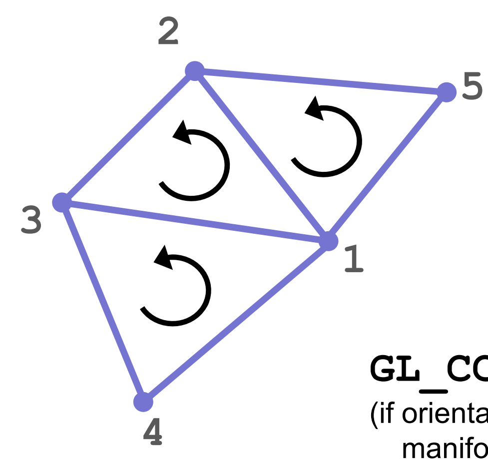
```
x1,y1,(z1)
x2,y2,(z2)
x3,y3,(z3)
x4,y4,(z4)

...
```

face list

```
1,2,3
1,3,4
2,1,5

...
```



`GL_CCW`

(if orientable manifold)

Less redundancy, more efficient in terms of memory

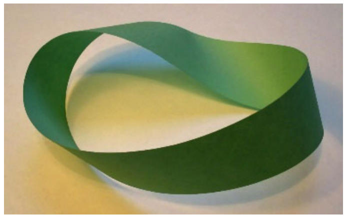
Easy to change vertex positions; still have to do (global) search for shared edges (local information)

Orientability (2-manifold embedded in 3D)

Orientability of 2-manifold:

Possible to assign consistent normal vector orientation

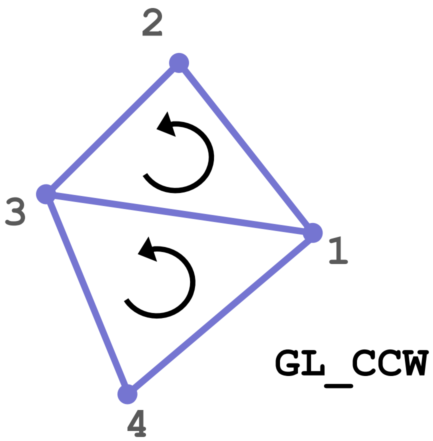
not orientable



Moebius strip
(only one side!)

Triangle meshes

- Edges
  - Consistent ordering of vertices: CCW (counter-clockwise) or CW (clockwise) (e.g., (3,1,2) on one side of edge, (1,3,4) on the other side)
- Triangles
  - Consistent front side vs. back side
  - Normal vector; or ordering of vertices (CCW/CW)
  - See also: “right-hand rule”

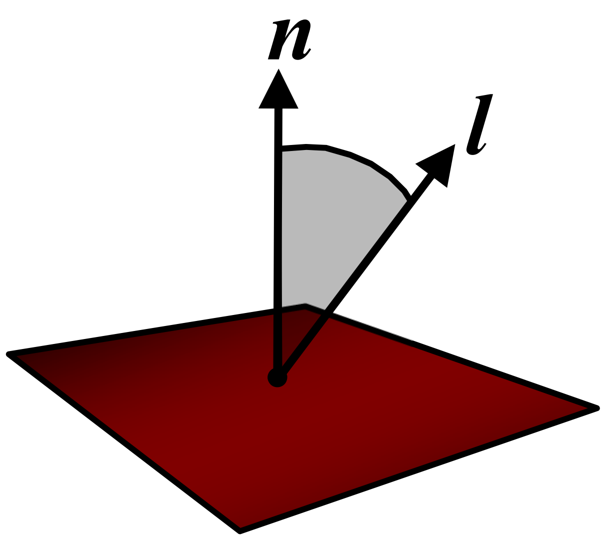


Local Shading Equations

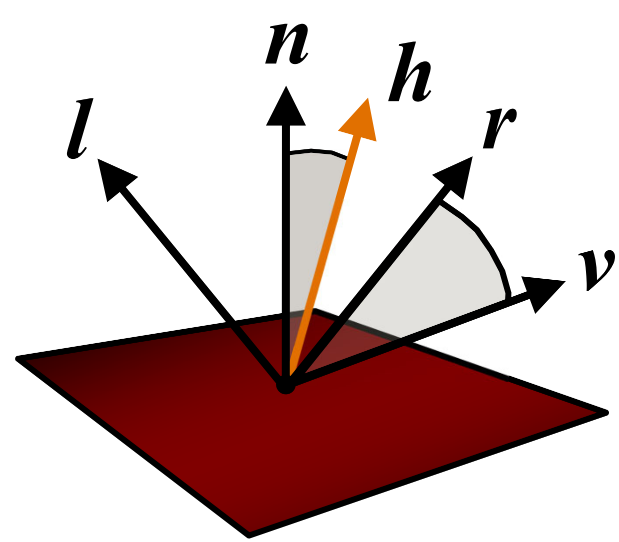
Standard volume shading adapts surface shading

Most commonly Blinn/Phong model

But what about the "surface" normal vector?



diffuse reflection



specular reflection

Thank you.

Thanks for material

- Helwig Hauser
- Eduard Gröller
- Daniel Weiskopf
- Torsten Möller
- Ronny Peikert
- Philipp Muigg
- Christof Rezk-Salama