King Abdullah University of Science and Technology KAUST

CS 247 – Scientific Visualization
Lecture 11: Scalar Field Visualization, Pt. 5

Markus Hadwiger, KAUST

Reading Assignment #6 (until Mar 12)

Read (required):

- Real-Time Volume Graphics, Chapter 2
  (*GPU Programming*)
- Real-Time Volume Graphics, Chapters 5.5 and 5.6 (you already had to read - 5.4)
  (*Local Volume Illumination*)
- Refresh your memory on eigenvectors and eigenvalues:
  `https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors`

Look at (optional):

- Riemannian Geometry for Scientific Visualization (notes and videos [part 1])
  `https://vccvisualization.org/RiemannianGeometryTutorial/`

From 2D to 3D (Domain)

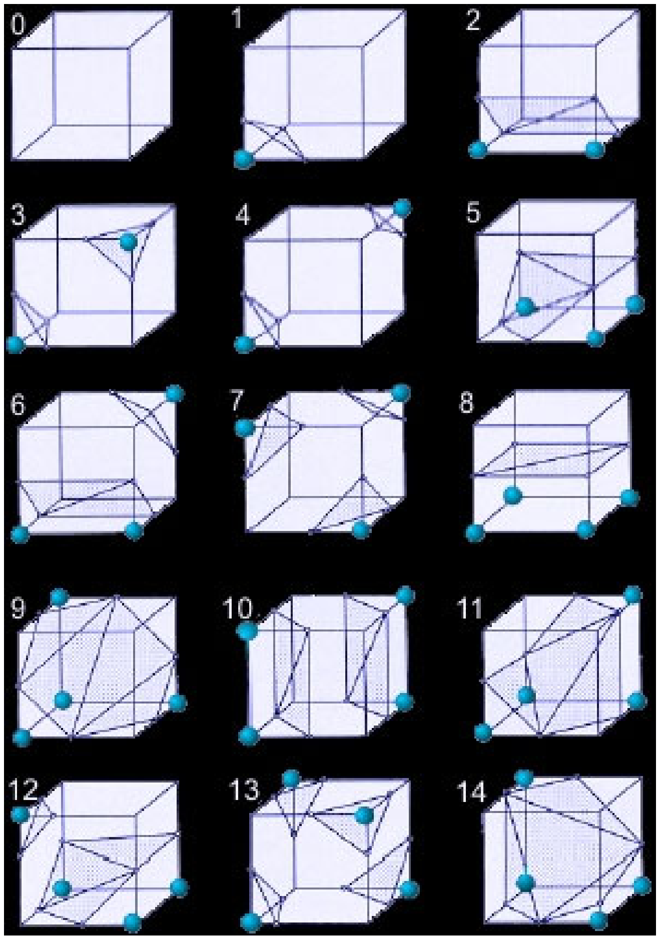
2D - Marching Squares Algorithm:

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:

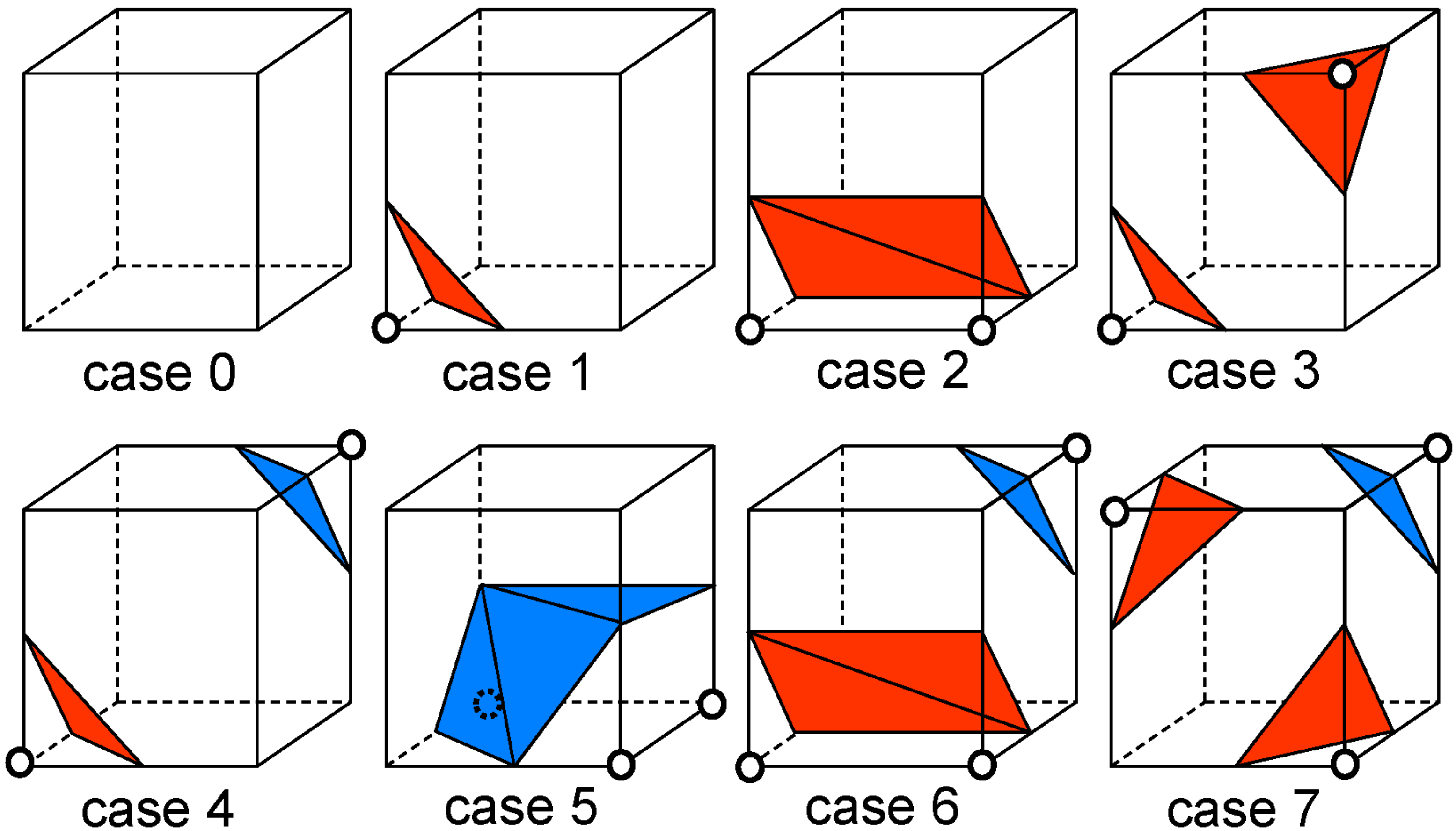
- rotational symmetries of the cube
- reflective symmetries of the cube
- sign changes of $\tilde{f}(x_i)$

They published a reduced set of 14*) cases shown on the next slides where

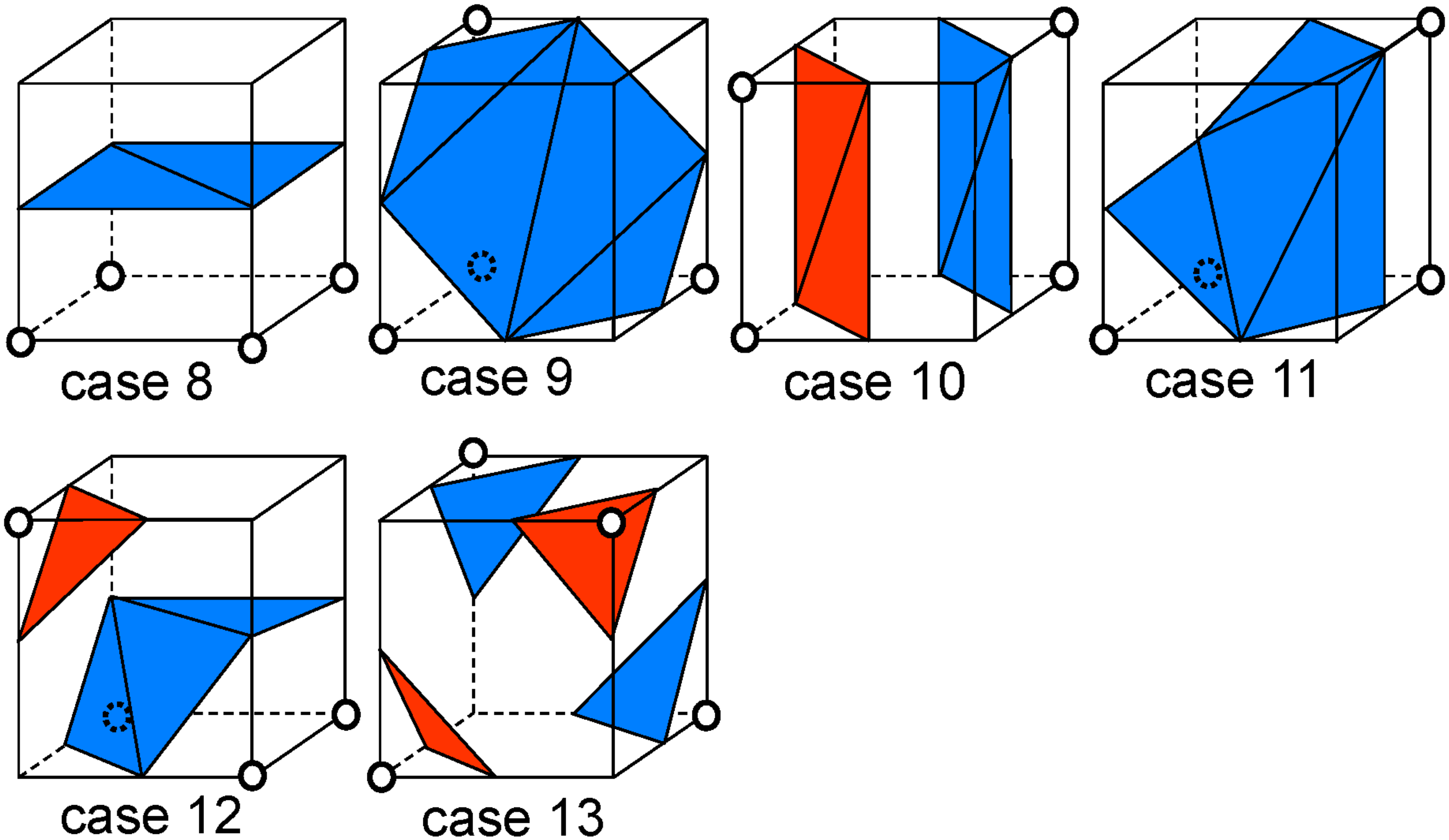
- white circles indicate positive signs of $\tilde{f}(x_i)$
- the positive side of the isosurface is drawn in red, the negative side in blue.

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

The marching cubes algorithm



The marching cubes algorithm



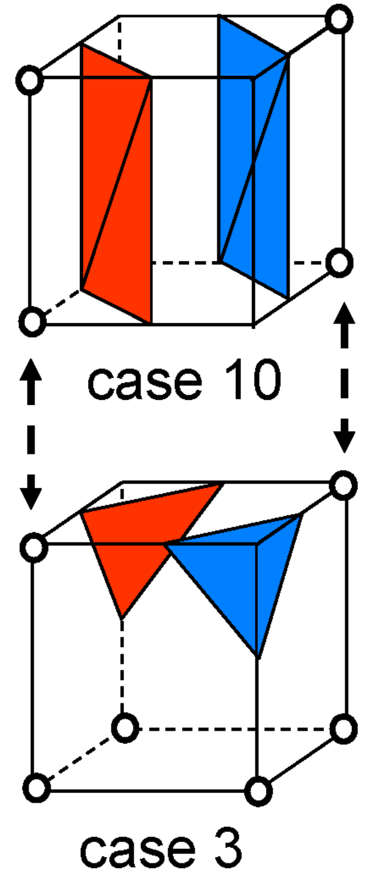
Do the pieces fit together?

- 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.

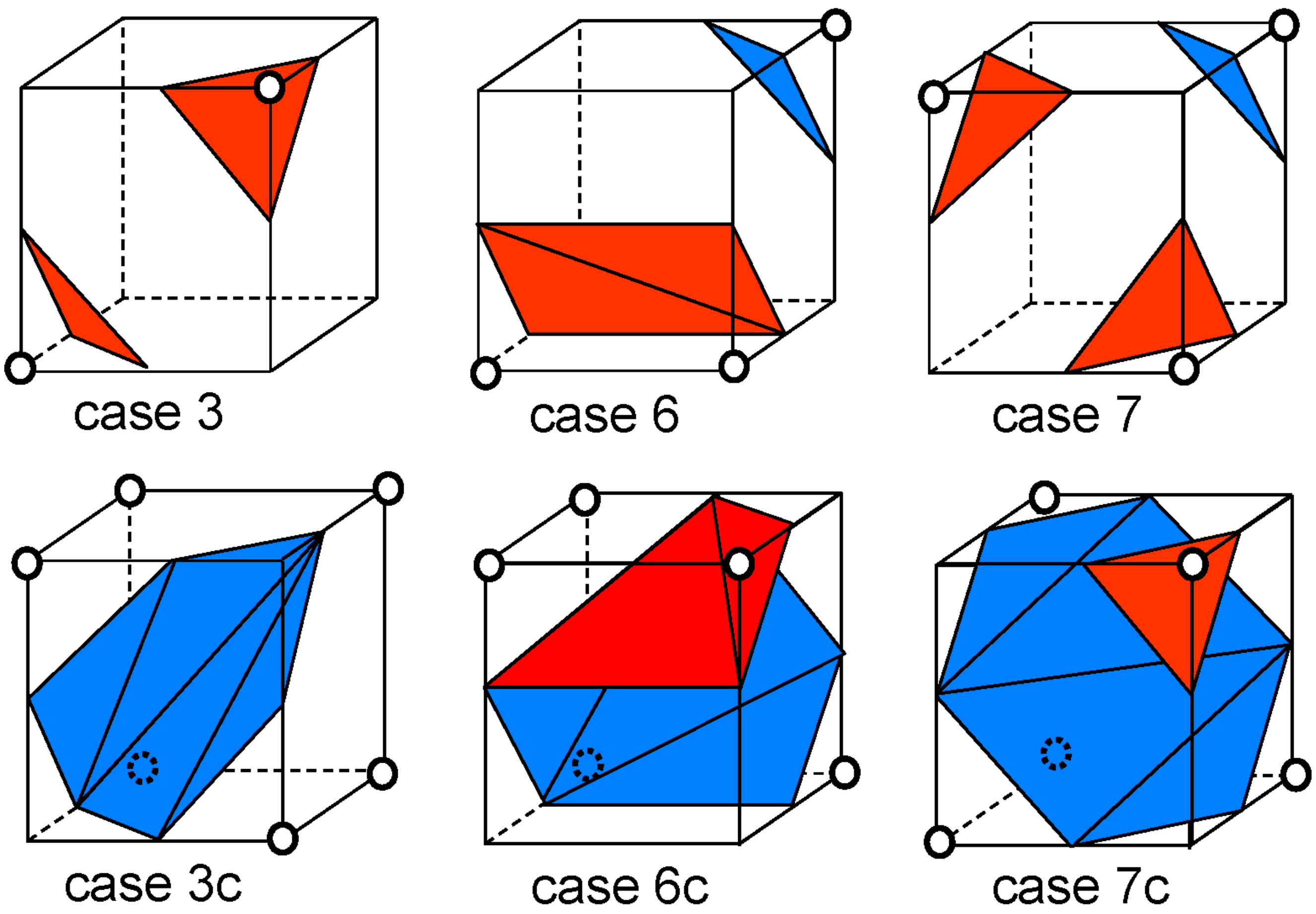
Example

- 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)

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

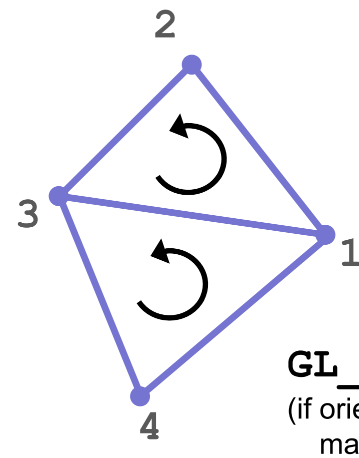
Render, e.g., with OpenGL immediate mode, …

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

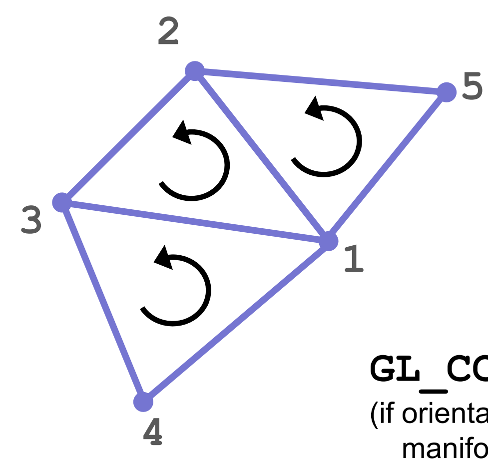
Triangle Mesh Data Structure (2)

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	face list
`x1,y1,(z1)`	`1,2,3`
`x2,y2,(z2)`	`1,3,4`
`x3,y3,(z3)`	`2,1,5`
`x4,y4,(z4)`	...
...	



`GL_CCW`

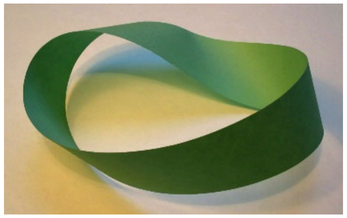
(if orientable manifold)

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)

not orientable



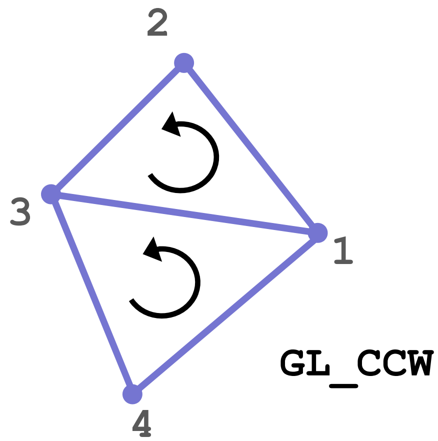
Moebius strip
(only one side!)

Orientability of 2-manifold:

Possible to assign consistent normal vector orientation

Triangle meshes

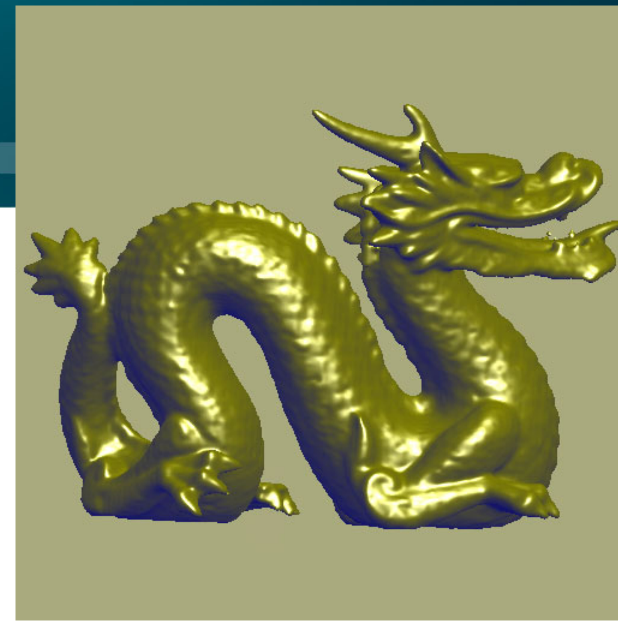
- Edges
  - Consistent ordering of vertices: CCW (counter-clockwise) or CW (clockwise)
    (e.g., (3,1,2) on one side of edge, (1,3,4) on the other side)
- Triangles
  - Consistent front side vs. back side
  - Normal vector; or ordering of vertices (CCW/CW)
  - See also: “right-hand rule”



Iso-Surface / Volume Illumination

What About Volume Illumination?

Crucial for perceiving shape and depth relationships



this is a scalar volume (3D distance field)!

Local Illumination in Volumes

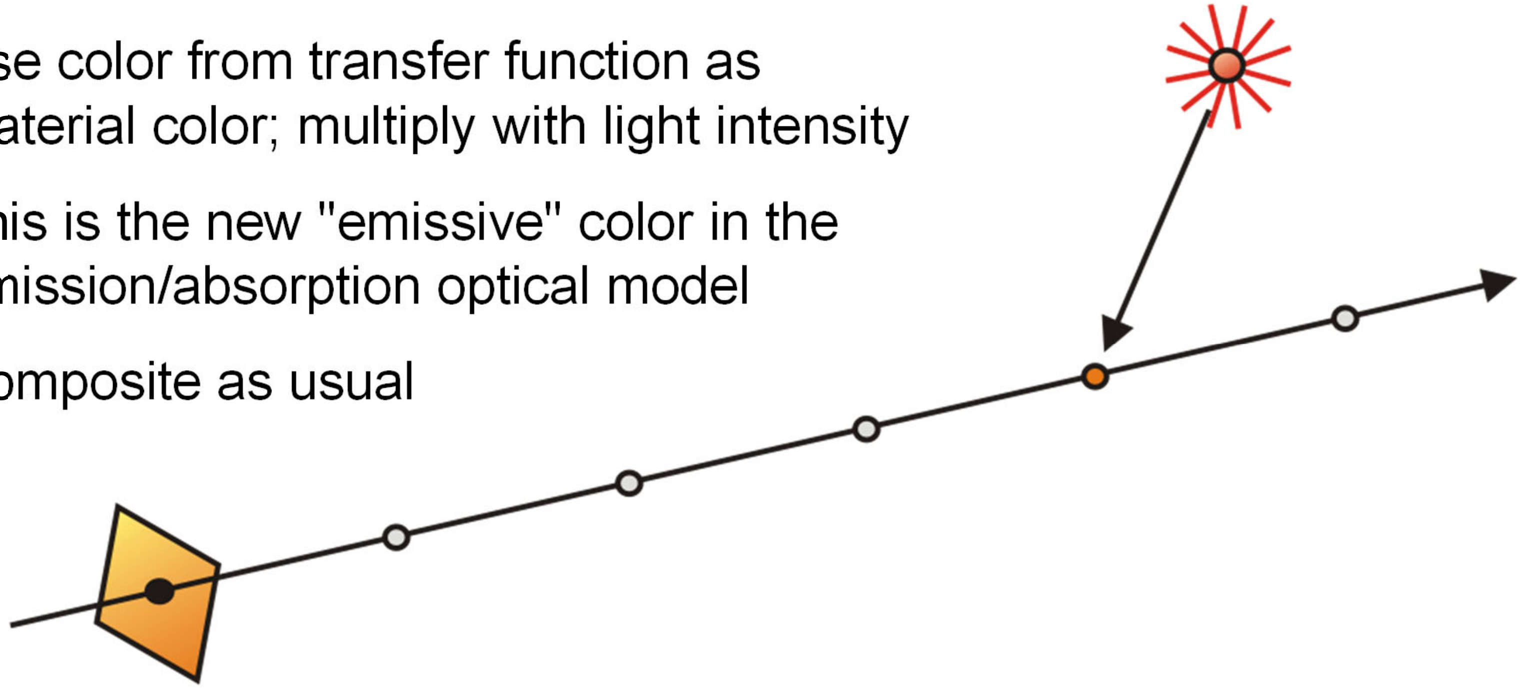
Interaction between light source and point in the volume

Local shading equation; evaluate at each point along a ray

Use color from transfer function as material color; multiply with light intensity

This is the new "emissive" color in the emission/absorption optical model

Composite as usual



Local Illumination in Volumes

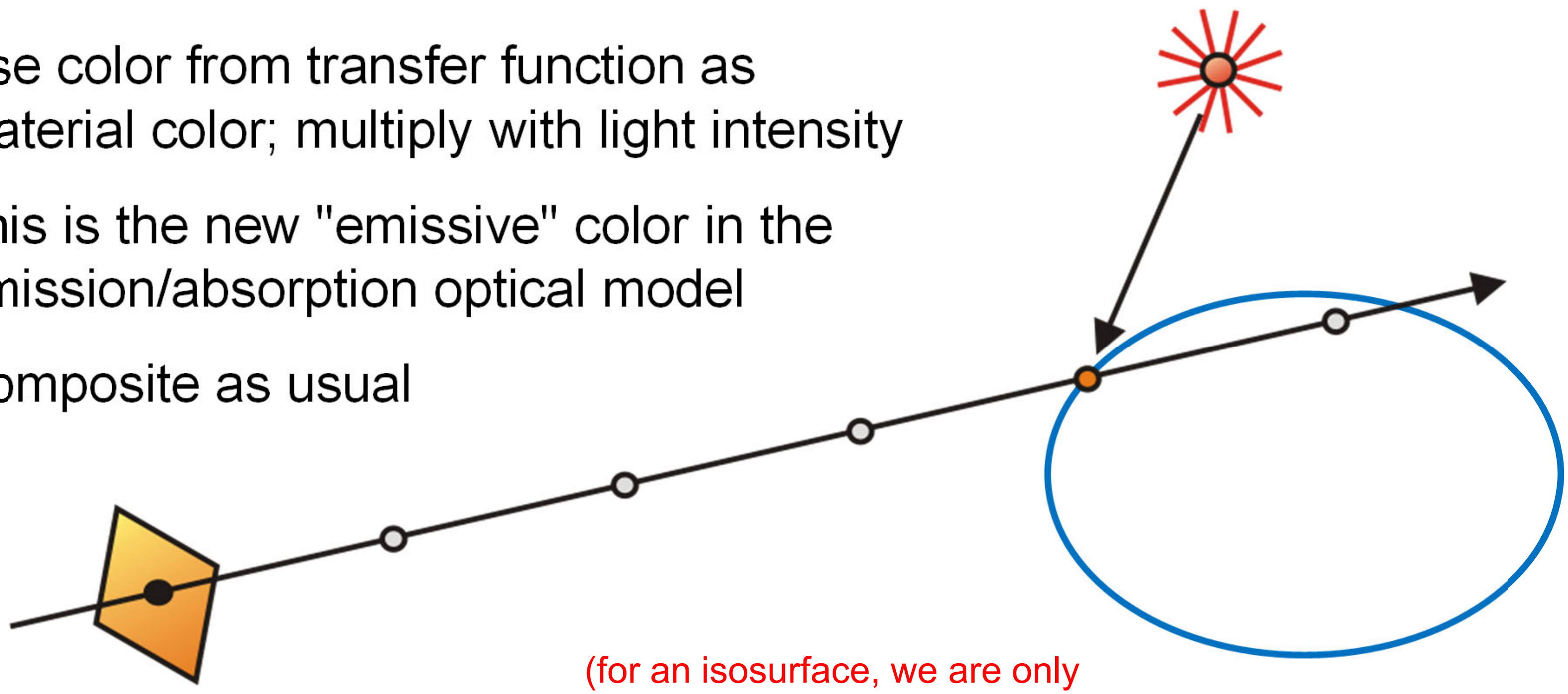
Interaction between light source and point in the volume

Local shading equation; evaluate at each point along a ray

Use color from transfer function as material color; multiply with light intensity

This is the new "emissive" color in the emission/absorption optical model

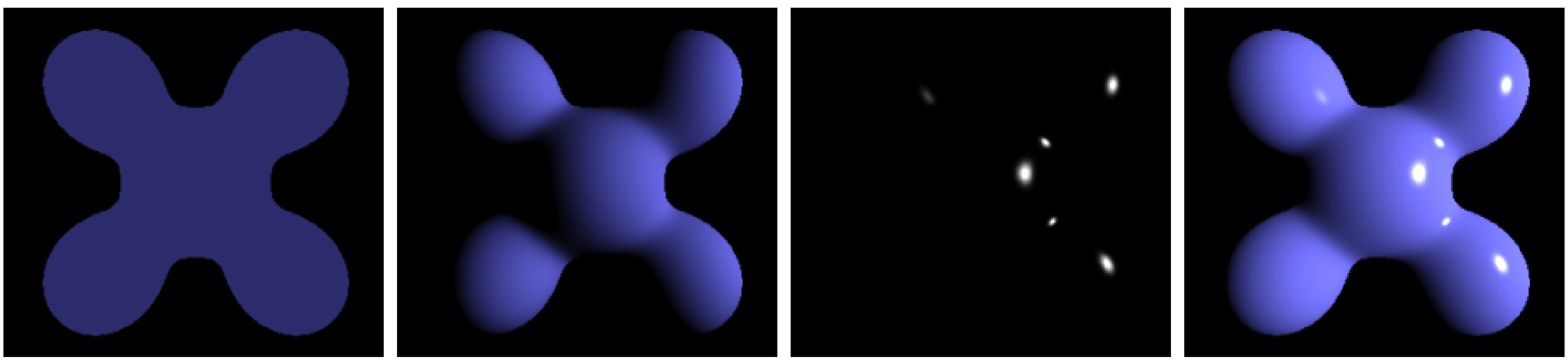
Composite as usual



(for an isosurface, we are only interested in points *on* the surface; in marching cubes: the vertices)

Local Illumination Model: Phong Lighting Model

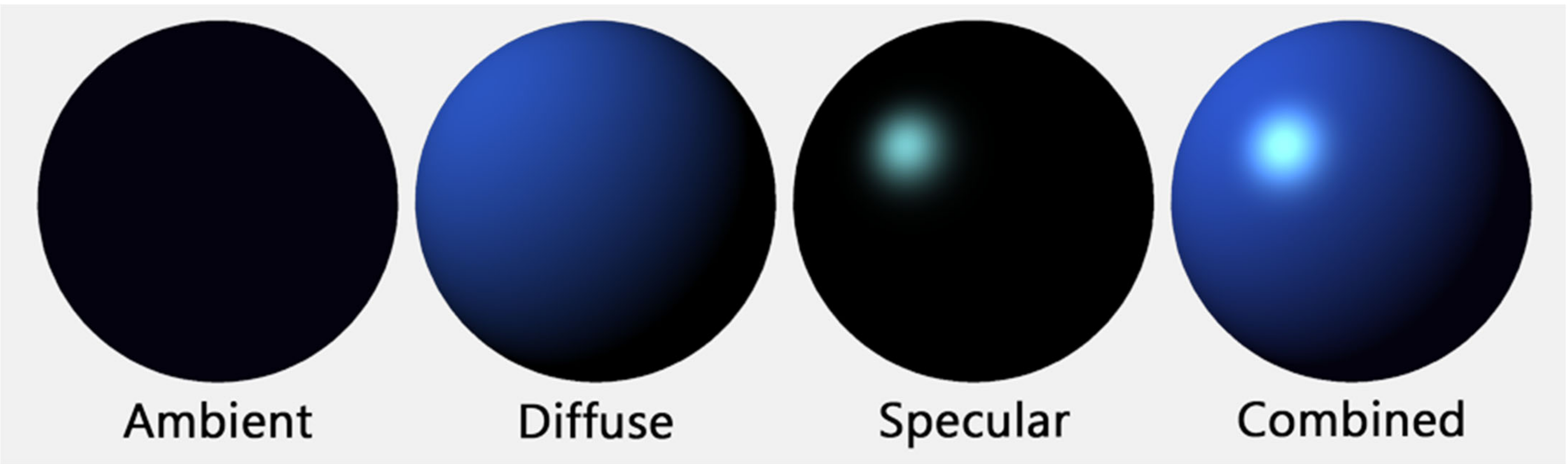
$$\mathbf{I}_{\text{Phong}} = \mathbf{I}_{\text{ambient}} + \mathbf{I}_{\text{diffuse}} + \mathbf{I}_{\text{specular}}$$



Ambient + Diffuse + Specular = Phong Reflection

Local Illumination Model: Phong Lighting Model

$$\mathbf{I}_{\text{Phong}} = \mathbf{I}_{\text{ambient}} + \mathbf{I}_{\text{diffuse}} + \mathbf{I}_{\text{specular}}$$

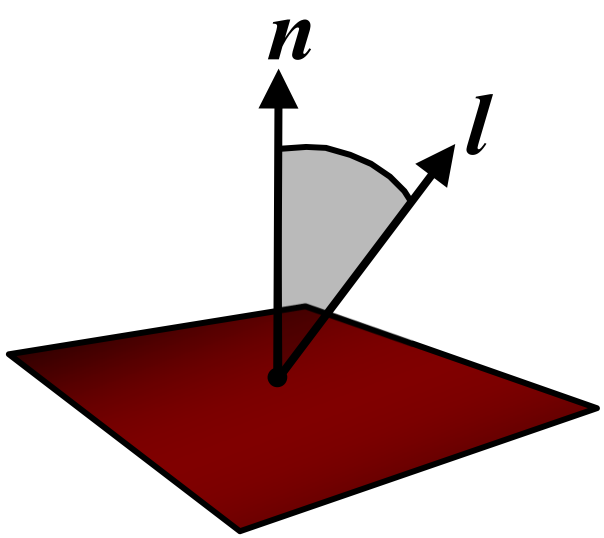


Local Shading Equations

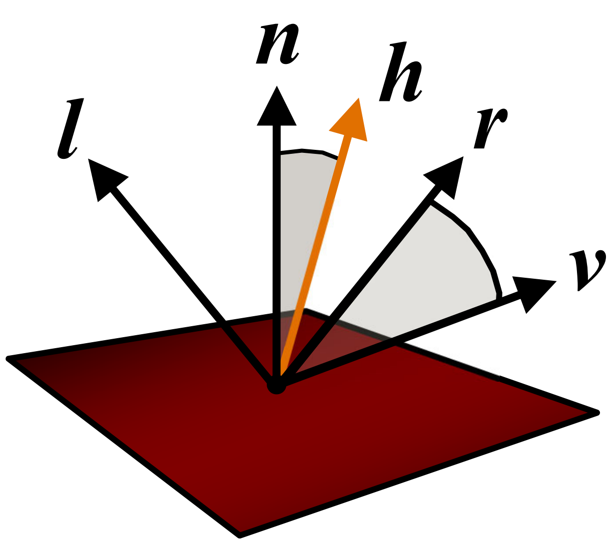
Standard volume shading adapts surface shading

Most commonly Blinn/Phong model

But what about the "surface" normal vector?



diffuse reflection



specular reflection

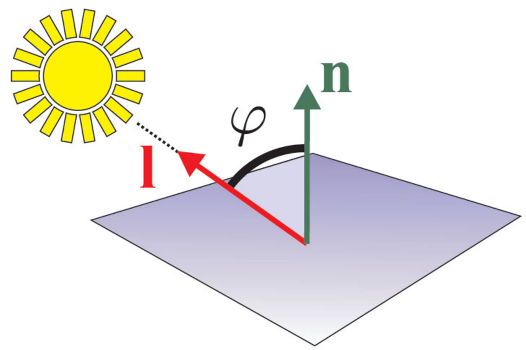
Local Illumination Model: Phong Lighting Model

$$\mathbf{I}_{\text{Phong}} = \mathbf{I}_{\text{ambient}} + \mathbf{I}_{\text{diffuse}} + \mathbf{I}_{\text{specular}}$$

$$\mathbf{I}_{\text{ambient}} = k_a \, \mathbf{M}_a \, \mathbf{I}_a$$

Local Illumination Model: Phong Lighting Model

$$\mathbf{I}_{\text{Phong}} = \mathbf{I}_{\text{ambient}} + \mathbf{I}_{\text{diffuse}} + \mathbf{I}_{\text{specular}}$$



$$\begin{aligned}\mathbf{I}_{\text{diffuse}} &= k_d\,\mathbf{M}_d\,\mathbf{I}_d \cos\varphi \qquad \text{if } \varphi \le \frac{\pi}{2} \\ &= k_d\,\mathbf{M}_d\,\mathbf{I}_d \max((\mathbf{n}\cdot\mathbf{l}), 0)\end{aligned}$$

The Dot Product (Scalar / Inner Product)

Cosine of angle between two vectors times their lengths

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos\theta$$

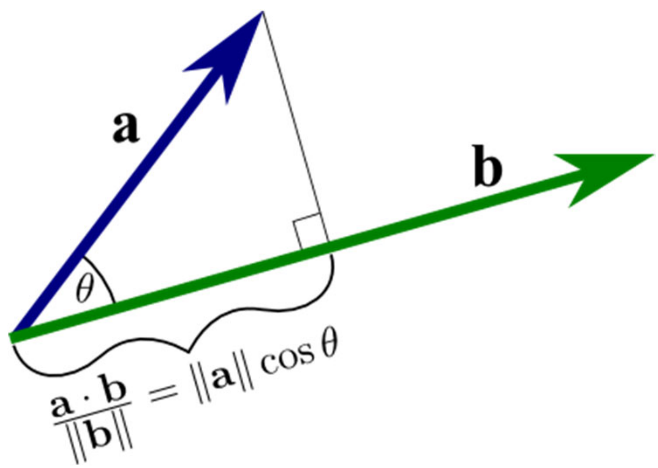
(geometric definition,
independent of coordinates)

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

(standard inner product
in Cartesian coordinates)

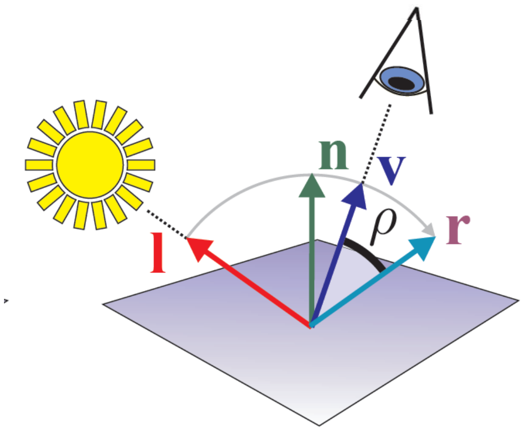
Many uses:

- Project vector onto another vector
- Project into basis (using the dual basis, see later)
- Project into tangent plane



Local Illumination Model: Phong Lighting Model

$$\mathbf{I}_{\text{Phong}} = \mathbf{I}_{\text{ambient}} + \mathbf{I}_{\text{diffuse}} + \mathbf{I}_{\text{specular}}$$

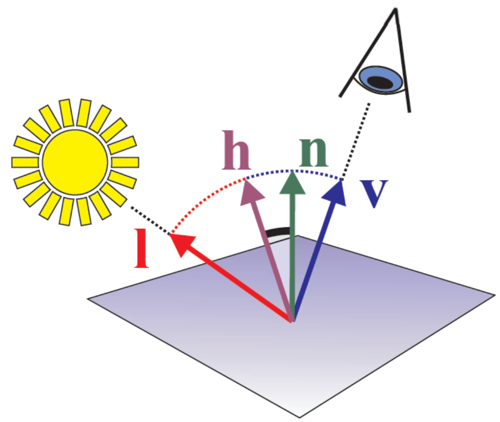


$$\begin{aligned}\mathbf{I}_{\text{specular}} &= k_s\,\mathbf{M}_s\,\mathbf{I}_s\cos^n\rho\,, \quad \text{if } \rho \le \frac{\pi}{2} \\ &= k_s\,\mathbf{M}_s\,\mathbf{I}_s\,(\mathbf{r}\cdot\mathbf{v})^n\end{aligned}$$

must also clamp!

Local Illumination Model: Phong Lighting Model

$$\mathbf{I}_{\text{Phong}} = \mathbf{I}_{\text{ambient}} + \mathbf{I}_{\text{diffuse}} + \mathbf{I}_{\text{specular}}$$



$$\mathbf{I}_{\text{specular}} \approx k_s\,\mathbf{M}_s\,\mathbf{I}_s\,(\mathbf{h}\cdot\mathbf{n})^n$$

must also clamp!

$$\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$$

half-way vector

The Gradient as Normal Vector

Gradient of the scalar field gives direction+magnitude of fastest change

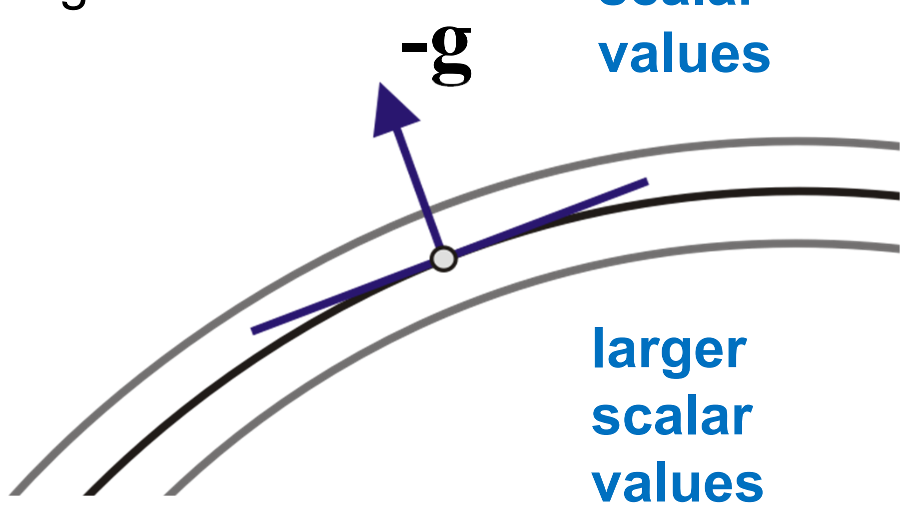
$$\mathbf{g} = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^{\mathbf{T}}$$

(only correct in Cartesian coordinates: see later)

Local approximation to isosurface at any point:
tangent plane = plane orthogonal to gradient

Normal of this isosurface:
normalized gradient vector
(negation is common convention)

$$\mathbf{n} = -\mathbf{g}/|\mathbf{g}|$$



Gradient and Directional Derivative

Gradient $\nabla f(x,y,z)$ of scalar function $f(x,y,z)$:

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

(only correct in Cartesian coordinates: see later)

Directional derivative in direction $\mathbf{u}$:

$$D_{\mathbf{u}} f(x,y,z) = \nabla f(x,y,z) \cdot \mathbf{u}$$

And therefore also:

$$D_{\mathbf{u}} f(x,y,z) = ||\nabla f|| \; ||\mathbf{u}|| \; \cos\theta$$

Gradient and Directional Derivative

Gradient $\nabla f(x,y,z)$ of scalar function $f(x,y,z)$:

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

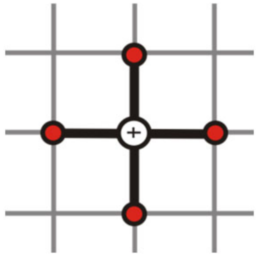
(only correct in Cartesian coordinates: see later)

(Cartesian vector components; basis vectors not shown)

But: always need **basis vectors**! With Cartesian basis:

$$\nabla f(x,y,z) = \frac{\partial f(x,y,z)}{\partial x}\,\mathbf{i} + \frac{\partial f(x,y,z)}{\partial y}\,\mathbf{j} + \frac{\partial f(x,y,z)}{\partial z}\,\mathbf{k}$$

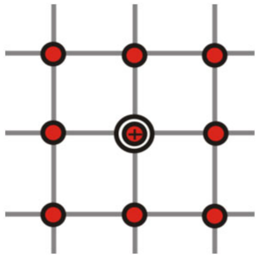
(Numerical) Gradient Reconstruction



We need to reconstruct the derivatives of a continuous function given as discrete samples

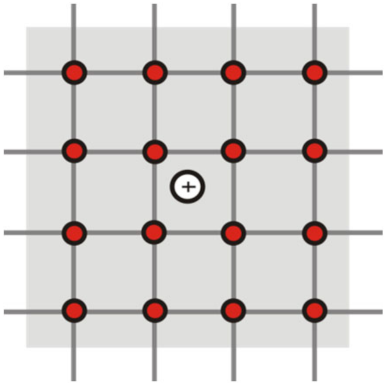
Central differences

- Cheap and quality often sufficient (2*3 neighbors in 3D)



Discrete convolution filters on grid

- Image processing filters; e.g. Sobel (3^3 neighbors in 3D)



Continuous convolution filters

- Derived continuous reconstruction filters
- E.g., the cubic B-spline and its derivatives (4^3 neighbors)

Finite Differences

Obtain first derivative from Taylor expansion

$$
\begin{aligned}
f(x_0 + h) &= f(x_0) + \frac{f'(x_0)}{1!}h + \frac{f''(x_0)}{2!}h^2 + \ldots \\
&= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} h^n .
\end{aligned}
$$

Forward differences / backward differences

$$
\begin{aligned}
f(x_0)' &= \frac{f(x_0 + h) - f(x_0)}{h} + o(h) \\
f(x_0)' &= \frac{f(x_0) - f(x_0 - h)}{h} + o(h)
\end{aligned}
$$

Finite Differences

Central differences

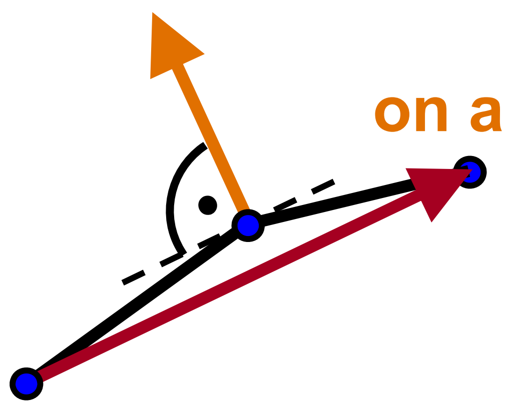
$$
\begin{aligned}
f(x_0 + h) &= f(x_0) + \frac{f'(x_0)}{1!} h + \frac{f''(x_0)}{2!} h^2 + o(h^3) \\
f(x_0 - h) &= f(x_0) - \frac{f'(x_0)}{1!} h + \frac{f''(x_0)}{2!} h^2 + o(h^3)
\end{aligned}
$$

$$
f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + o(h^2)
$$

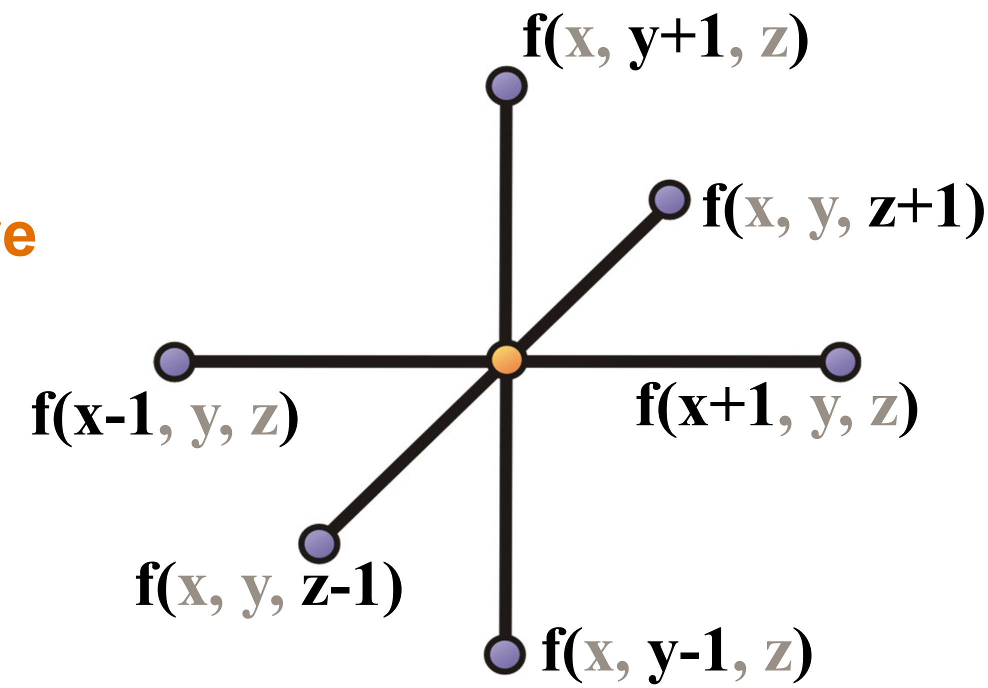
Central Differences

Need only two neighboring voxels per derivative

Most common method



on a curve




```
gx = 0.5( f(x+1, y, z) - f(x-1, y, z) )
gy = 0.5( f(x, y+1, z) - f(x, y-1, z) )
gz = 0.5( f(x, y, z+1) - f(x, y, z-1) )
```

in a volume

Thank you.

Thanks for material

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