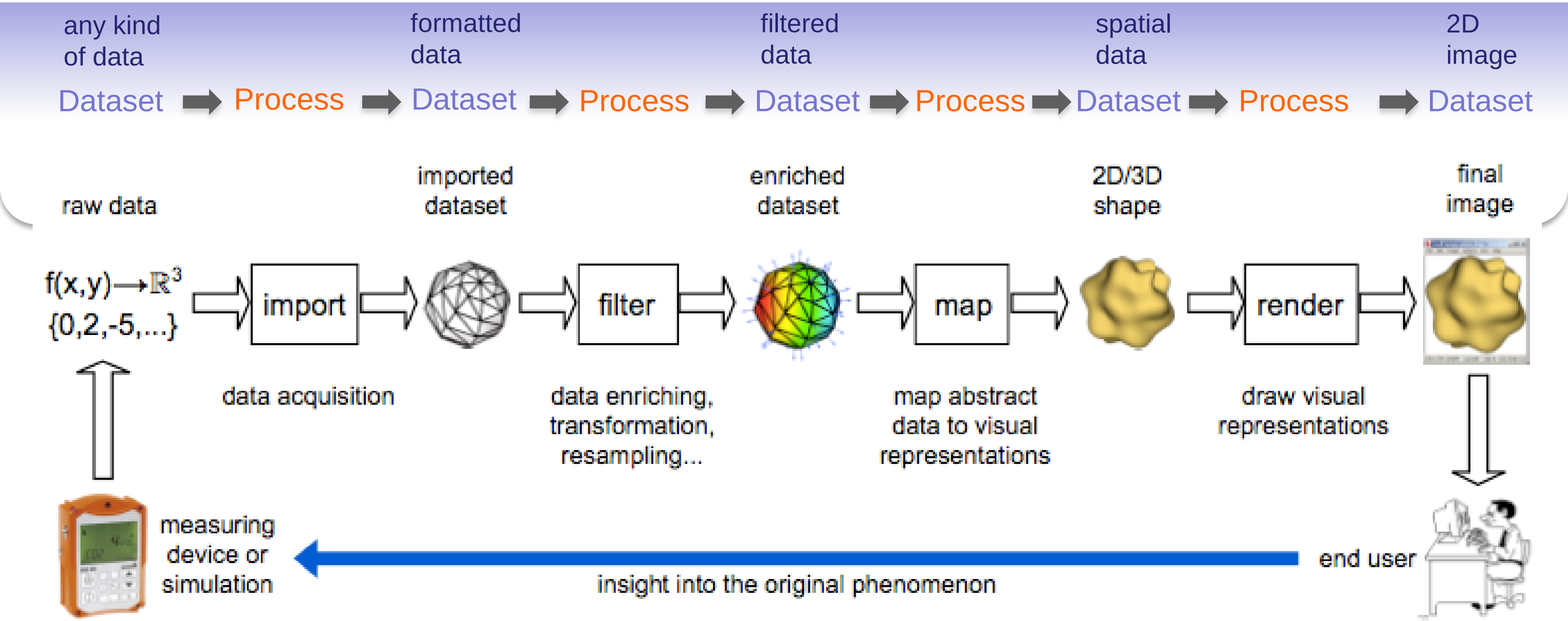
Data Representation

Cmpt 767 Visualization

Steven Bergner
sbergner@sfu.ca

[based on slides by A. C. Telea]

The Visualization Pipeline - Recall



[A.C Telea: Data Visualization, Principles and Practice, 2nd edition, CRC Press, 2014]

The Visualization Pipeline - Recall

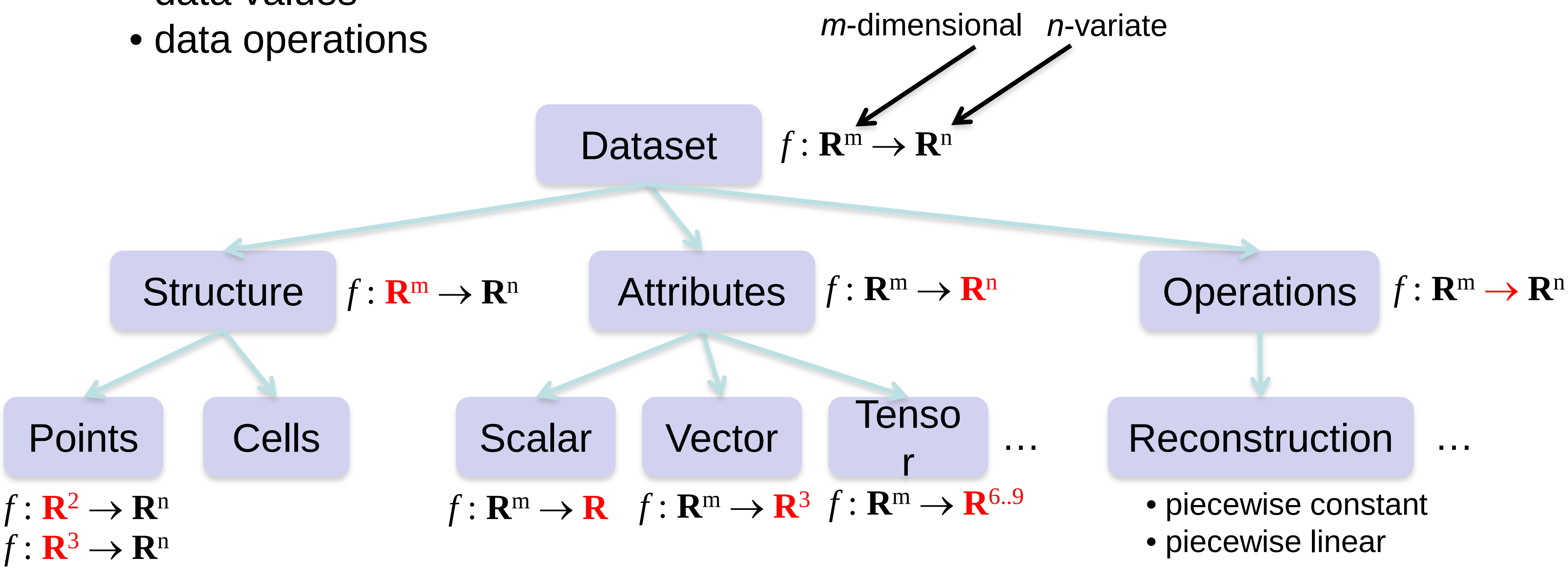
any kind of data		formatted data		filtered data		spatial data		2D image
Dataset →	Input →	Dataset →	Filtering →	Dataset →	Mapping →	Dataset →	Rendering →	Dataset

1. **Input data**
   - your primary “raw” source of information
   - can be anything (measurements, simulations, databases, …)
2. **Formatted data**
   - converted to points, cells, attributes (discussed next in this module)
   - Ready to use for visualization algorithms
3. **Filtered data**
   - eliminates the unneeded **data**, adds the needed **information**
   - read and written by visualization algorithms
4. **Spatial (mapped) data**
   - has spatial embedding → can be **drawn**
5. **2D Image**
   - final image you look at to get your answers

Scientific Visualization - The Dataset

Dataset

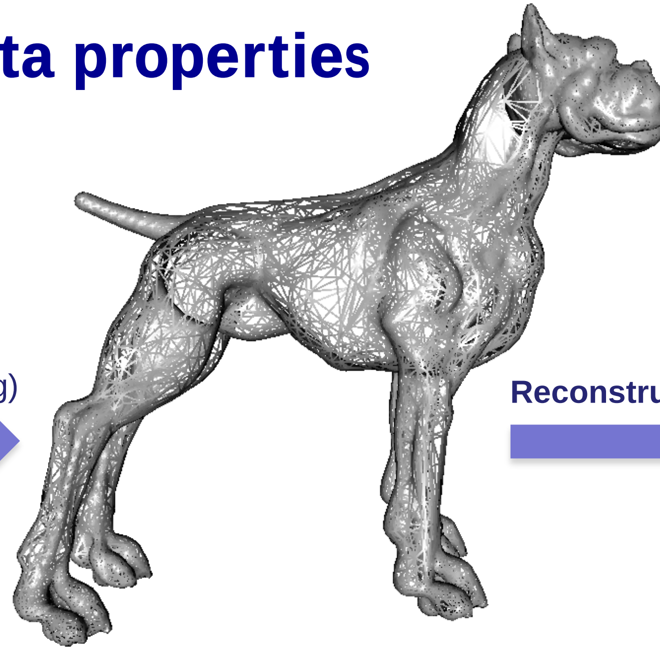
- key notion in visualization (SciVis, InfoVis, SoftVis)
- captures all relevant characteristics of a data collection
    - structure
    - data values
    - data operations



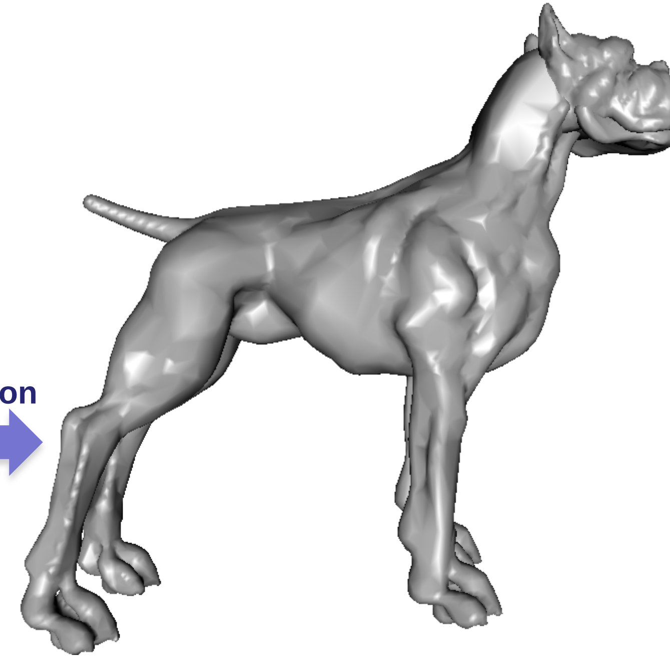
We'll detail all these next

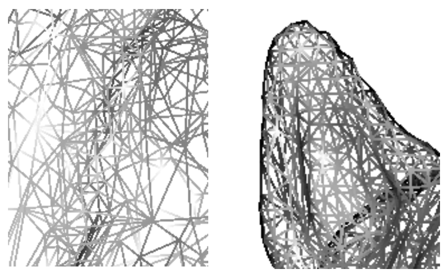
Visualization data properties

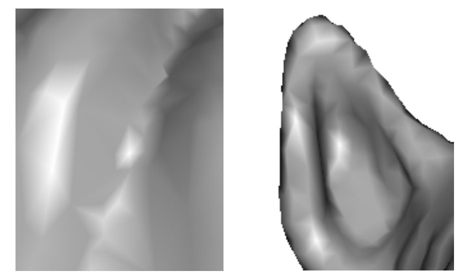
Sampling
(data importing)



Reconstruction







$f : \mathrm{D} \rightarrow \mathrm{C}$

Continuous data

Sampling

$\{p_i, f_i\}$ $\quad p_i \in \mathrm{D}$, $\; f_i \in \mathrm{C}$

Measurements (samples)
at discrete set of points

Reconstruction

$\tilde{f} : \mathrm{D} \rightarrow \mathrm{C} \quad \tilde{f}(p_i) = f(p_i) = f_i$

Continuous data, as close as possible to input

Continuous data

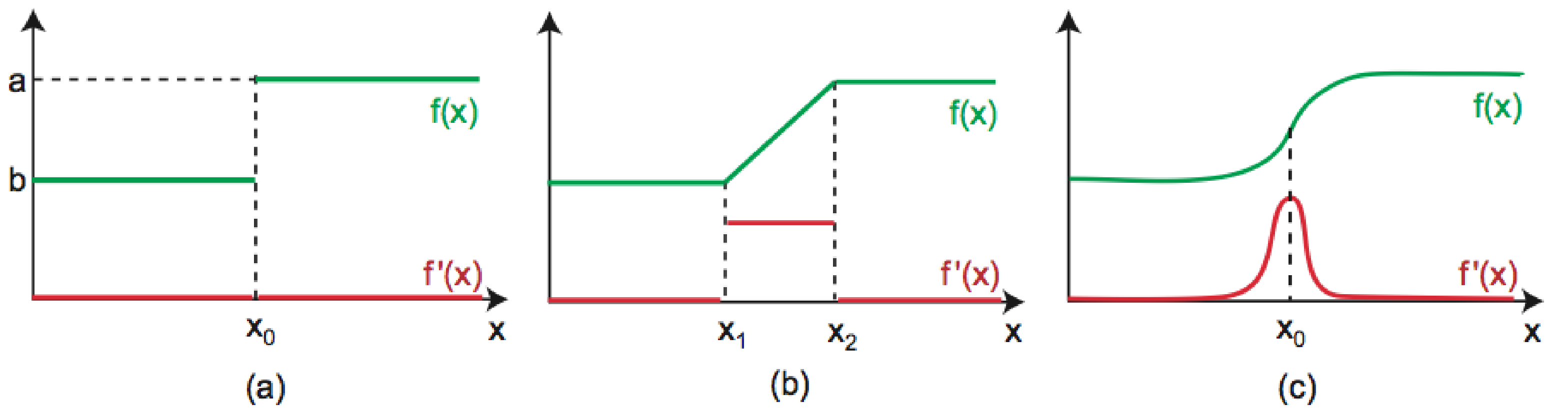


Figure 3.1. Function continuity. (a) Discontinuous function. (b) First-order C^0 continuous function. (c) High-order C^k continuous function.

Cauchy definition of continuity

A function f is continuous iff

$$\forall \epsilon > 0, \exists \delta > 0 \text{ such that if } \|x - p\| < \delta, x \in \mathrm{C} \text{ then } \|f(x) - f(p)\| < \epsilon.$$

C^{-1}	discontinuous (graph of function has “holes”)
C^{0}	first-order continuous (graph of function has “kinks”)
C^{k}	first k derivatives of the function are continuous

Sampled data

Functional properties

- **finite**
  - captures continuous signal at a finite set of points (measurements)
- **accurate**
  - can reconstruct a signal close to input accurately
  - reconstruction guarantees continuity properties

Non-functional properties

- **efficient**
  - reconstruction is fast
- **compact**
  - store Gbytes of sample points compactly
- **generic**
  - few data structures cover most dataset types
- **simple**
  - learn to create & use such data structures quickly

Interpolation

Fundamental tool for signal reconstruction

1. **Reconstruction** formula

$$\tilde{f} = \sum_{i=1}^{N} f_i \phi_i$$

$\phi_i : \mathrm{D} \to \mathrm{C}$ are basis (or interpolation) functions

2. **Interpolation**: reconstruction passes through (interpolates) the sampled values

$$\sum_{i=1}^{N} f_i \phi_i(p_j) = f_j, \forall j.$$

because $\tilde{f}(p_i) = f(p_i) = f_i$

3. **Orthogonality** of basis functions

$$\phi_i(p_j) = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

why? Just apply (2) to $f = \begin{cases} 1, p = p_j \\ 0, p \neq p_j \end{cases}$

4. **Normality** of basis functions

$$\sum_{i=1}^{N} \phi_i(x) = 1, \forall x \in \mathrm{D}$$

why? $\sum_{i=1}^{N} \phi_i(p_j) = 1, \forall p_j$ (sum (3) over $i = 1..N$)

and apply above to all $p_i \in D$

Practical interpolation: Cells

Recall the interpolation formula

$$\tilde{f} = \sum_{i=1}^{N} f_i \phi_i$$

This becomes very inefficient if

•N is very large and we have to evaluate ϕ_i at all these N points

ϕ_i have complicated expressions

Practical basis functions

•are non-zero over small spatial 'pieces' of D only (limited support)

•have the same simple formula at all sample points p_i

$\Rightarrow$ We will discretize our spatial domain D into **cells**

Cells: 1D space

Consider a simple 1D function $f : \mathbf{R} \to \mathbf{R}$

1. Sample the 1D axis at some points p_i
2. Define cells $c_i = (p_i, p_{i+1})$
3. Consider the reference basis functions for a reference cell $(0,1)$

   $\phi_{0,1} : [0,1] \to [0,1], \ \ \phi_0(r) = 1 - r, \ \ \phi_1(r) = r$
4. Define a linear transformation T_i from the reference to actual cell c_i

$$(x,y,z) = T(r,s,t) = \sum_{i=1}^{n} p_i \Phi_i^1(r,s,t)$$

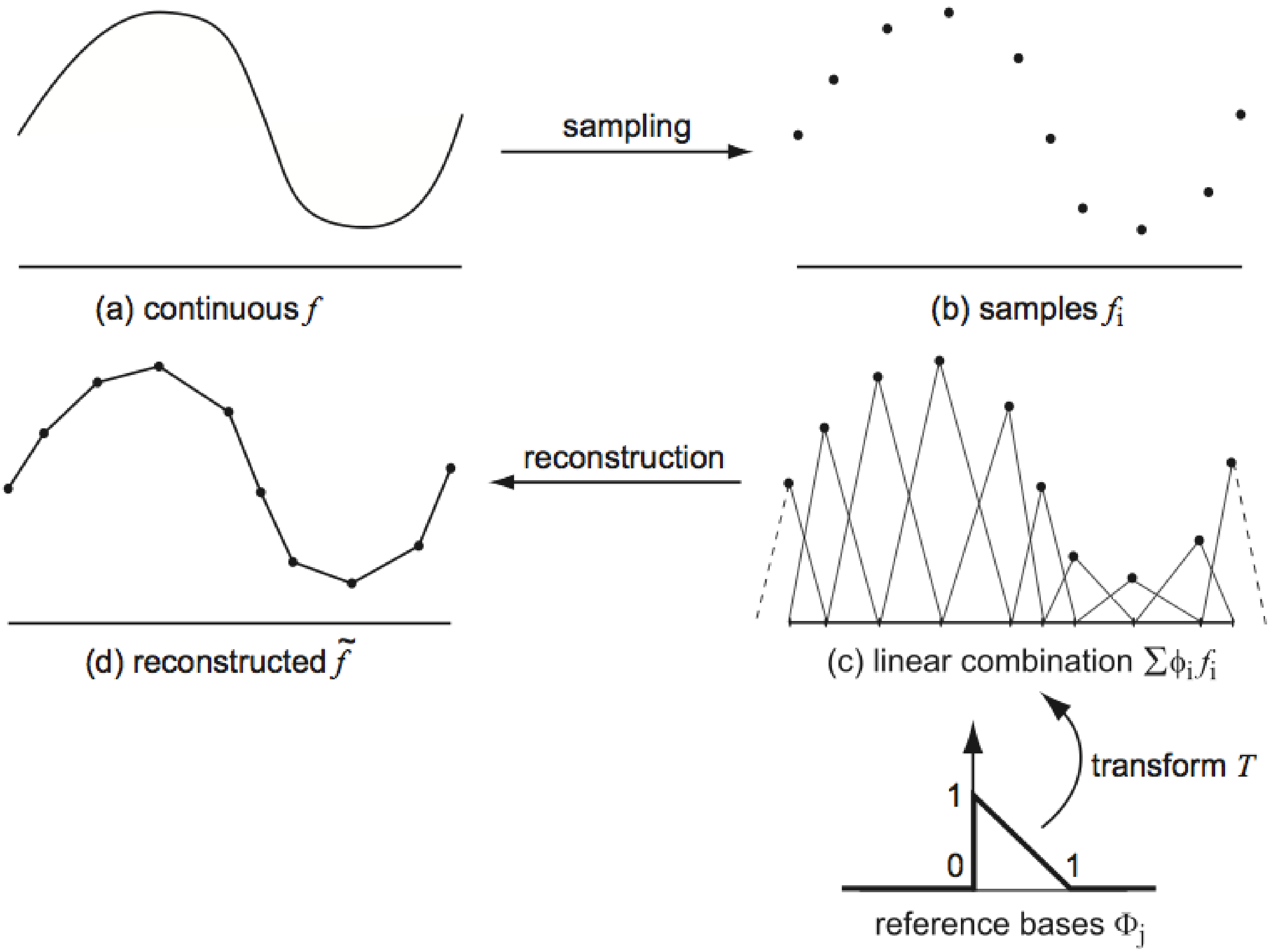
1. For c_i, define the actual basis functions $\Phi_{0,1}$ using $\phi_{0,1}$ and T_i^{-1} and rewrite the final interpolation

$$\tilde{f}(x,y) = \sum_{i=1} f_i \Phi_i^1(T^{-1}(x,y))$$

2. Apply (5) to interpolate all points in c_i using only samples at vertices p_i, p_{i+1} of c_i
3. Repeat from 4 for next cell c_{i+1}

Note: see Sec. 3.4 for expressions for all T^{-1}

Cells: 1D example (cont'd)



Remarks

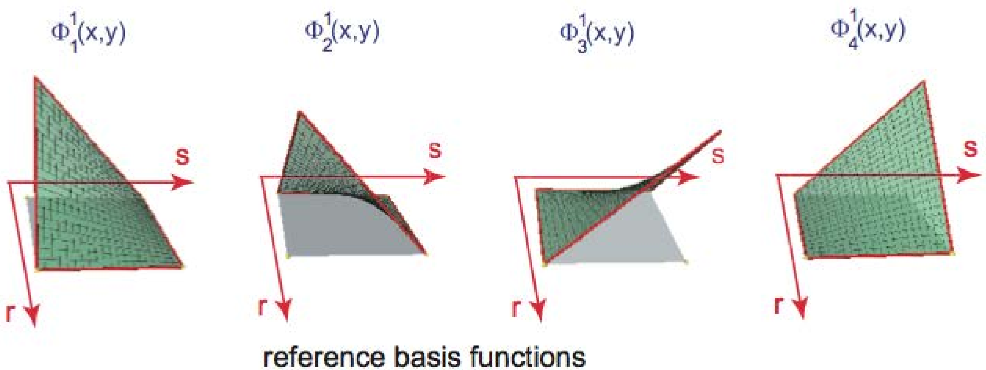
- interpolation & reconstruction goes cell-by-cell
- only need sample points at a cell vertices to interpolate over that cell
- reconstruction is C^1 because ϕ_i are C^1 and interpolation formula is are C^∞

2D cells: Quads

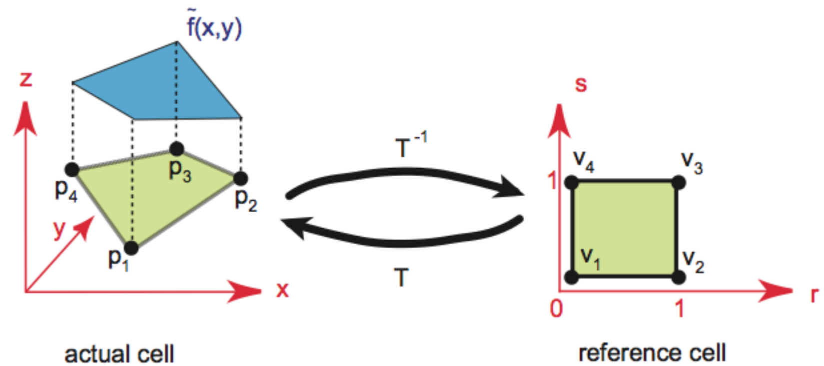
Same as in 1D case, but

- we have to decide on different cells; say we take quads
- quads $\rightarrow$ 4 vertices, 4 basis functions
- particular case: square cells = pixels

Bilinear basis functions



Bilinear transforms

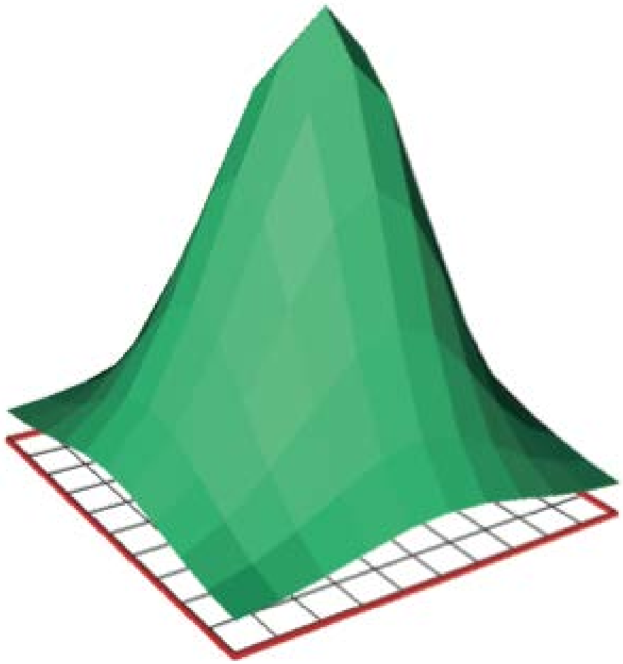


$$
\begin{aligned}
\Phi^1_1(r,s) &= (1-r)(1-s),\\
\Phi^1_2(r,s) &= r(1-s),\\
\Phi^1_3(r,s) &= rs,\\
\Phi^1_4(r,s) &= (1-r)s;
\end{aligned}
$$

See book, p 47-50

2D cells: Quads

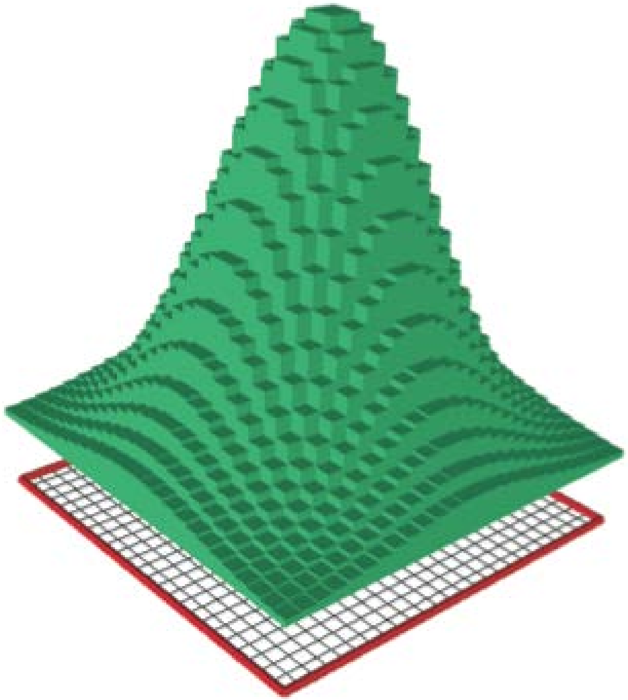
Bilinear interpolation



$$\begin{aligned}\Phi_1^1(r,s) &= (1-r)(1-s),\\ \Phi_2^1(r,s) &= r(1-s),\\ \Phi_3^1(r,s) &= rs,\\ \Phi_4^1(r,s) &= (1-r)s;\end{aligned}$$

- 4 functions, one per vertex
- result: C^0 but never C^1 (why?)
- good for vertex-based samples

Constant interpolation

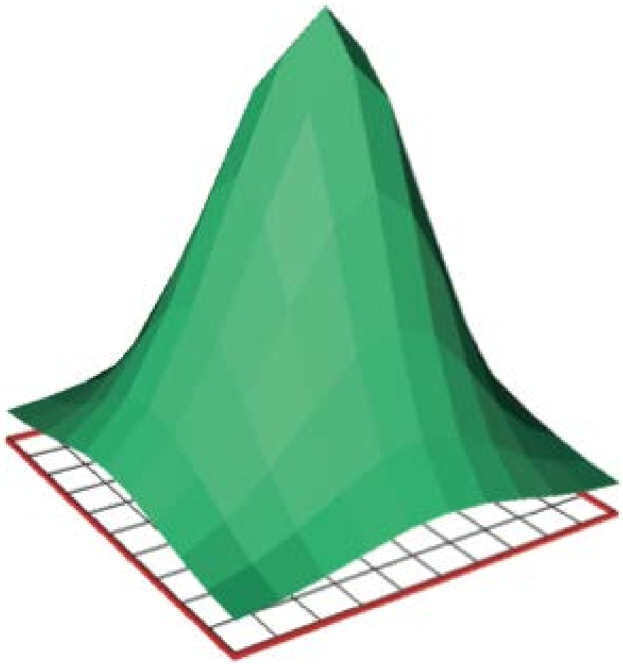


$$\phi_i^0(x) = \begin{cases} 1, & x \in c_i,\\ 0, & x \notin c_i.\end{cases}$$

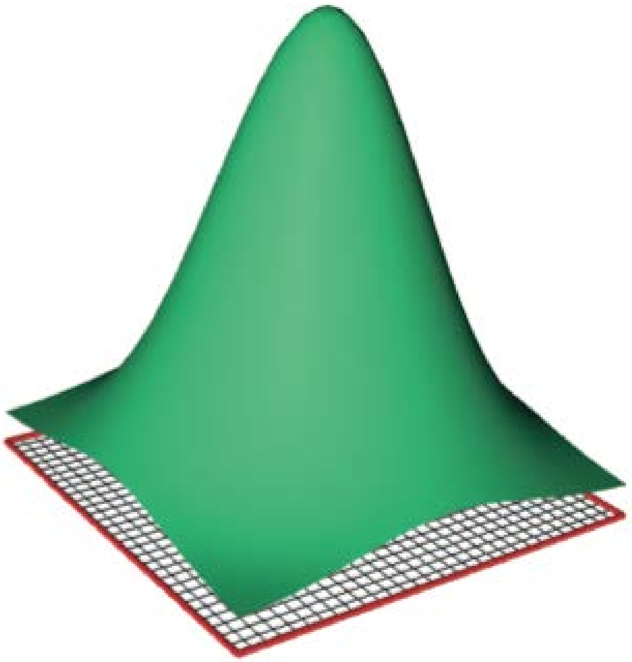
- 1 functions per whole cell
- result: not even C^0
- good for cell-based samples

Intermezzo

What is the difference between flat and Gouraud (smooth) shading?



Flat shading



Gouraud shading

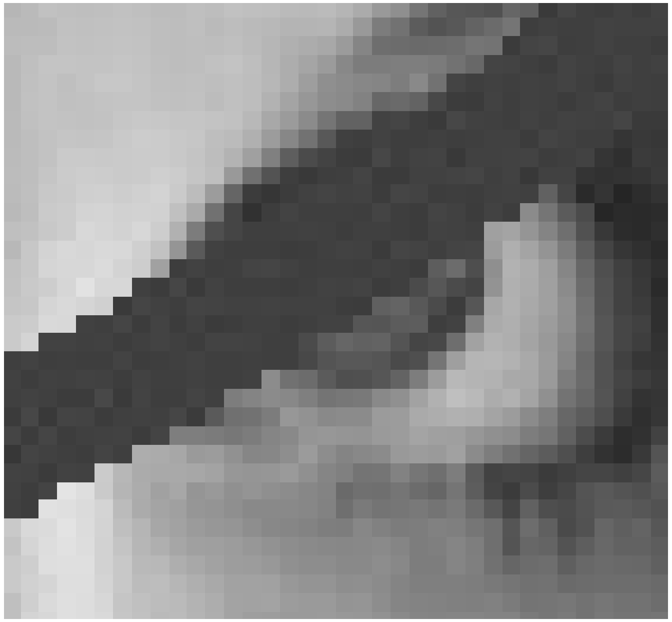
- surface: bilinear interpolation
- colors: constant interpolation

- surface: bilinear interpolation
- colors: bilinear interpolation

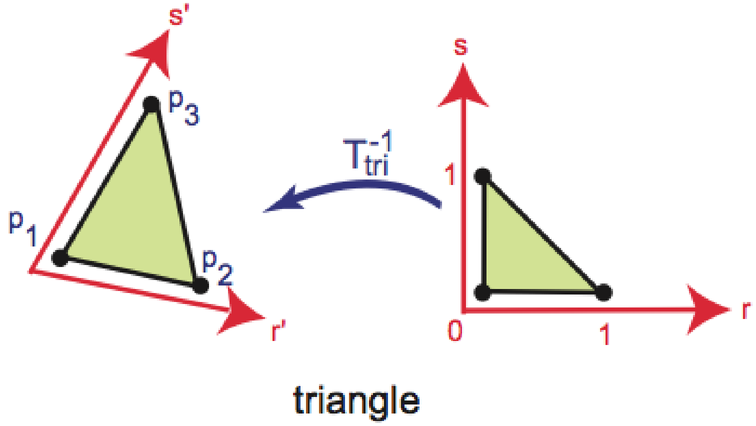
Note: do not confuse *Gouraud shading* (color interpolation) with the *Phong lighting model* (color computation from normals)

2D cells: Quads

Images (color or grayscale)
•use constant basis functions
•cells = pixels
•data (color) is defined at the center of pixels, not corners
•we’ll see why this is important in Module 3



2D cells: Triangles

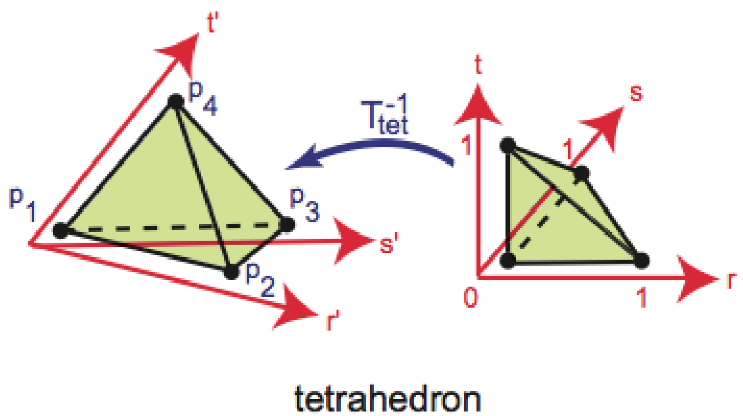


$$\Phi_1^1(r,s) = 1 - r - s,$$
$$\Phi_2^1(r,s) = r,$$
$$\Phi_3^1(r,s) = s.$$

Remarks

- triangles and quads offers largely same pro's and con's
- quad basis functions are not planes (they are **bi**linear)
- in graphics/visualization, triangles used more often than quads
  - easier to cover complex shapes with triangles than quads
  - same computational complexity

3D cells: Tetrahedra

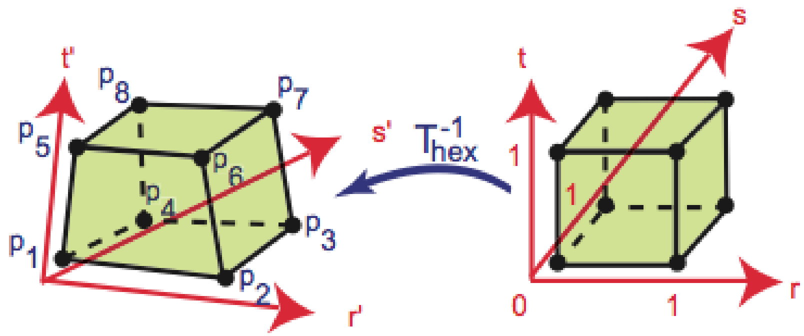


$$
\begin{aligned}
\Phi_1^1(r,s,t) &= 1-r-s-t,\\
\Phi_2^1(r,s,t) &= r,\\
\Phi_3^1(r,s,t) &= s,\\
\Phi_4^1(r,s,t) &= t.
\end{aligned}
$$

Remarks

- counterparts of triangles in 3D
- interpolate volumetric functions $f : \mathbf{R}^3 \to \mathbf{R}$
- three parametric coordinates r, s, t
- **tri**linear interpolation

3D cells: Hexahedra



hexahedron

$$\Phi_1^1(r,s,t) = (1-r)(1-s)(1-t),$$
$$\Phi_2^1(r,s,t) = r(1-s)(1-t),$$
$$\Phi_3^1(r,s,t) = rs(1-t),$$
$$\Phi_4^1(r,s,t) = (1-r)s(1-t),$$
$$\Phi_5^1(r,s,t) = (1-r)(1-s)t,$$
$$\Phi_6^1(r,s,t) = r(1-s)t,$$
$$\Phi_7^1(r,s,t) = rst,$$
$$\Phi_8^1(r,s,t) = (1-r)st.$$

Remarks

- counterparts of quads in 3D
- interpolate volumetric functions $f : \mathbf{R}^3 \to \mathbf{R}$
- **tri**linear interpolation
- particular case: cubic cells or voxels (studied later in Module 7)

Cell types for constant/linear basis functions

0D

•point

1D

•line

2D

•triangle, quad, rectangle

3D

•tetrahedron, parallelepiped, box, pyramid, prism, …

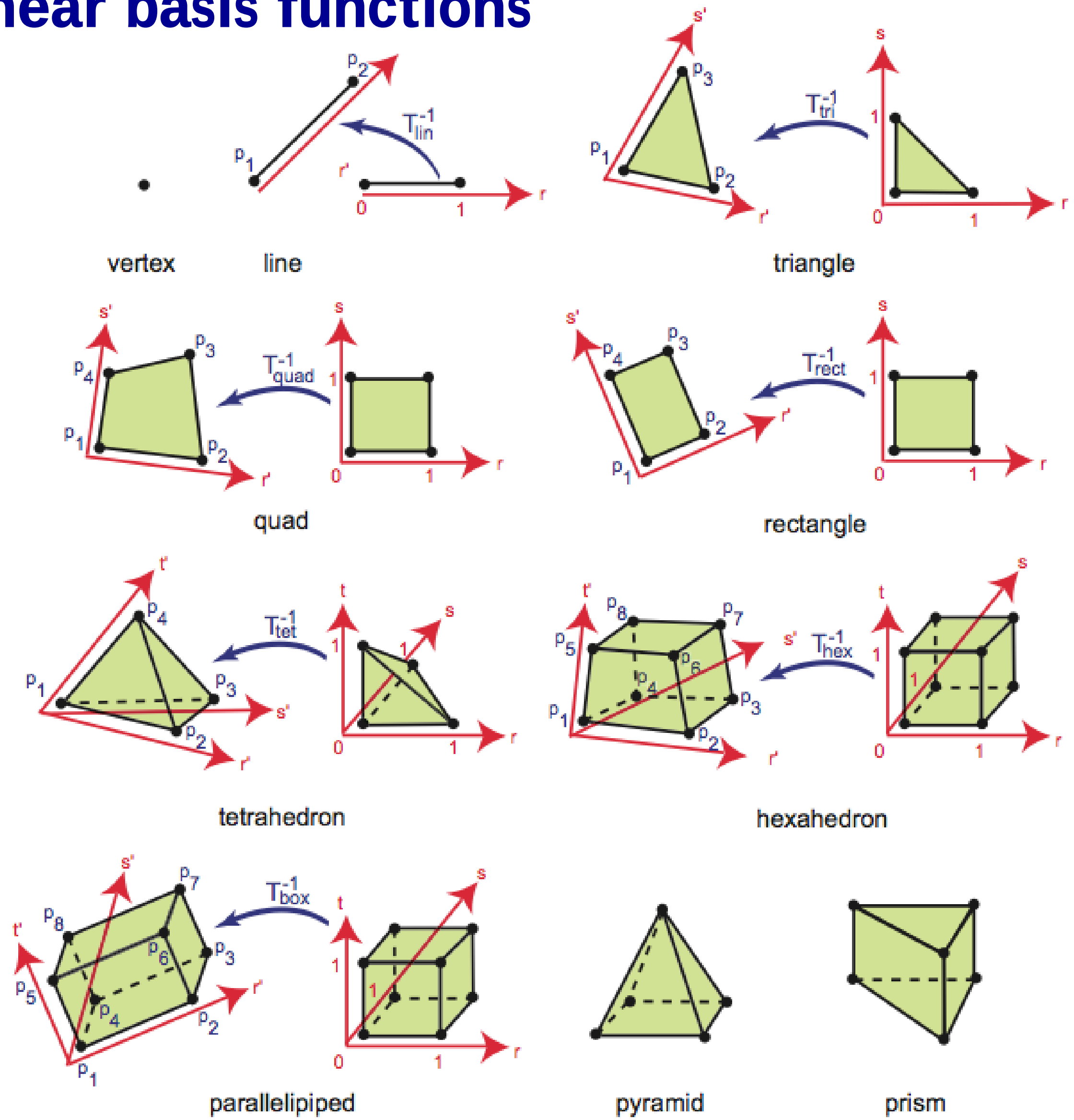


Figure 3.5. Cell types in world and reference coordinate systems.

Quadratic cells

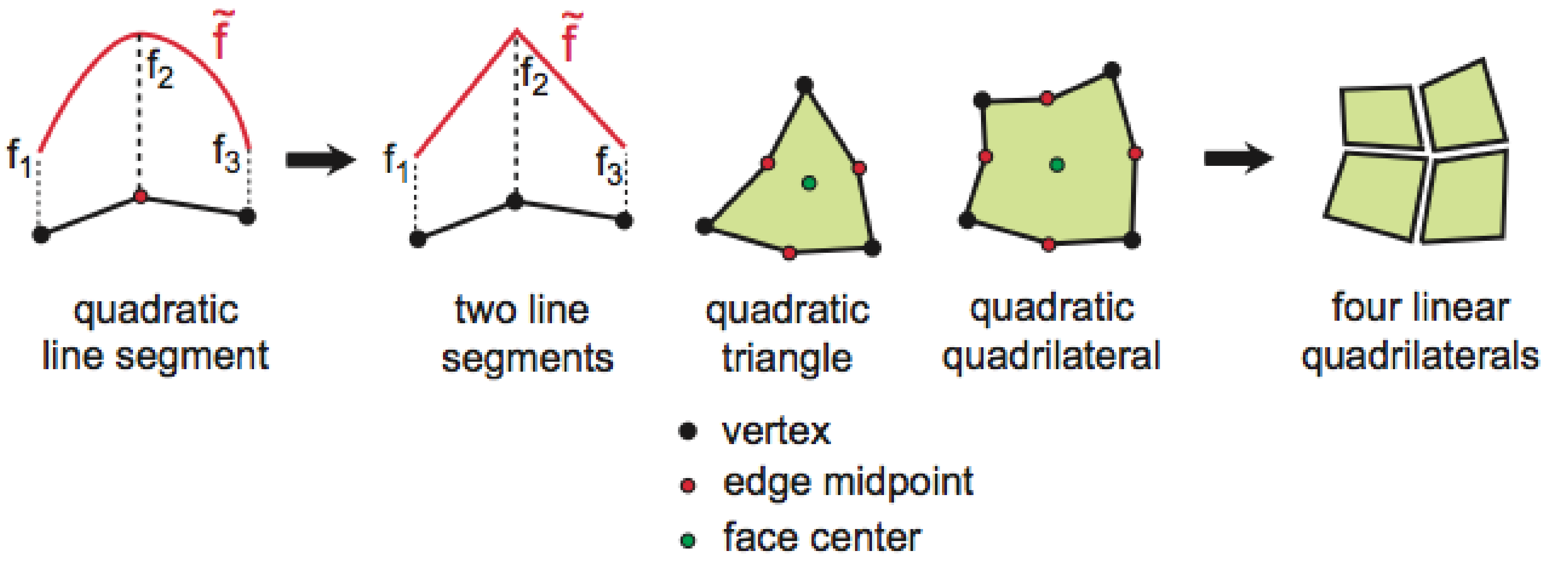


Figure 3.6. Converting quadratic cells to linear cells.

- allow defining quadratic basis functions
- higher precision for interpolation
- however, we need data samples at extra midpoints, not just vertices
- used in more complex numerical simulations (e.g. finite elements)
- split into linear cells for visualization purposes

From cells to grids

Cells

•provide interpolation over a small, simple-shaped spatial region

Grids

•partition our complex data domain D into cells

•allow applying per-cell interpolation (as described so far)

Given a domain D…

A grid G = {ci} is a set of cells such that

$$c_i \cap c_j = \emptyset, \forall i \neq j$$ no two cells overlap (why? Think about interpolation)

$$\bigcup_i c_i = D$$ the cells cover all our domain (why? Think about our end goal)

The dimension of the domain D constrains which cell types we can use: **see next**

Uniform grids

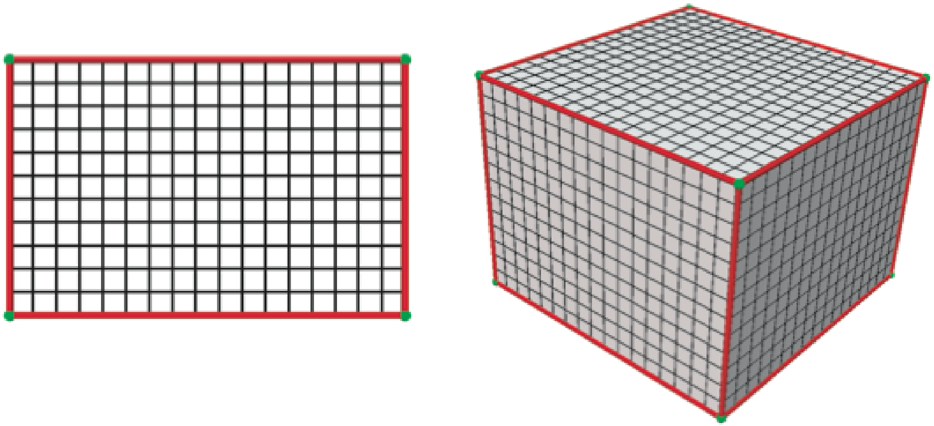
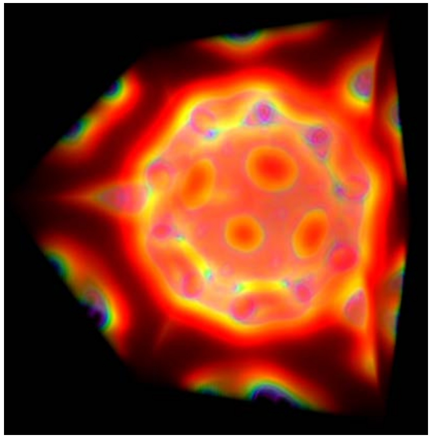


Figure 3.7. Uniform grids. 2D rectangular domain (left) and 3D box domain (right).

image



volume

- all cells have identical size and type (typically, square or cubic)
- cannot model non-axis-aligned domains

Storage requirements

- m integers for the #cells along each of the m dimensions of D (e.g. m=2 or 3)

Rectilinear grids

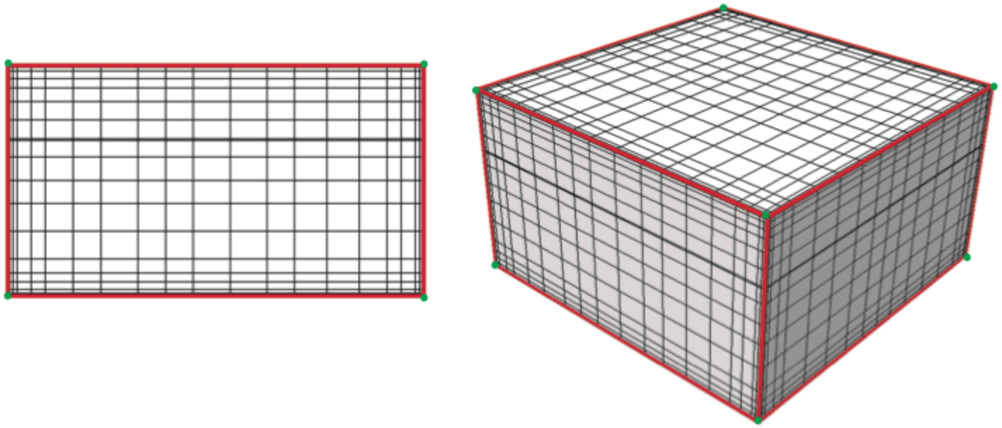


Figure 3.8. Rectilinear grids. 2D rectangular domain (left) and 3D box domain (right).

- all cells have same type
- cells can have different dimensions but share them along axes
- cannot model non-axis-aligned domains

Storage requirements

$\sum_{i=1}^{m} d_i$ floats (coordinates of vertices along each of the m axes of D)

Structured grids

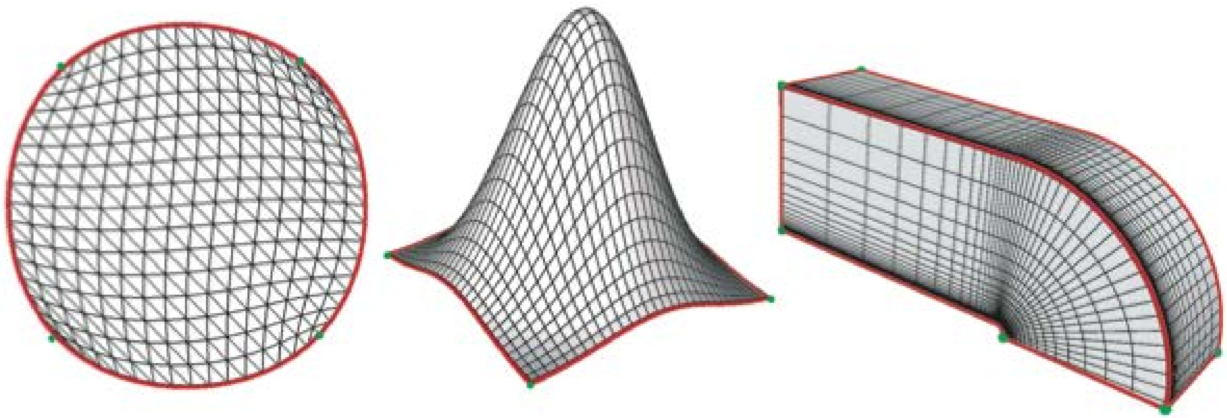


Figure 3.9. Structured grids. Circular domain (left), curved surface (middle), and 3D volume (right). Structured grid edges and corners are drawn in red and green, respectively.

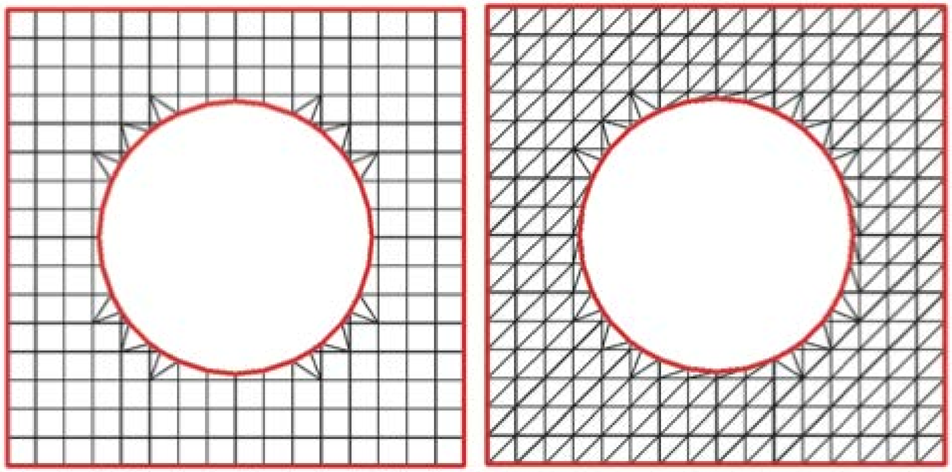
•all cells have same type
•cell vertex coordinates are freely (explicitly) specifiable…
•…as long as cells assemble in a matrix-like structure
•can approximate more complex shapes than rectilinear/uniform grids

Storage requirements

$\prod_{i=1}^{m} d_i$ floats (coordinates of all vertices)

Unstructured grids

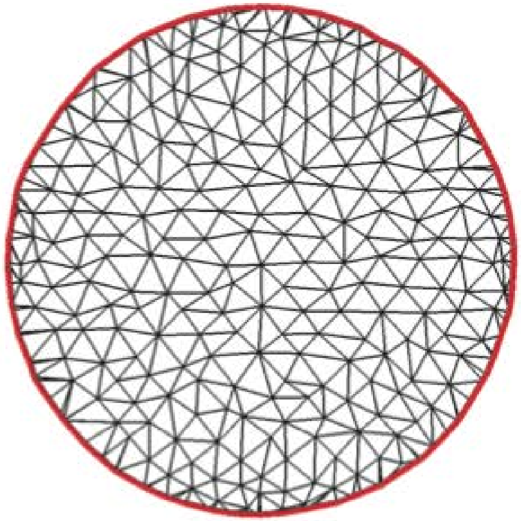
Consider the domain D: a square with a hole in the middle

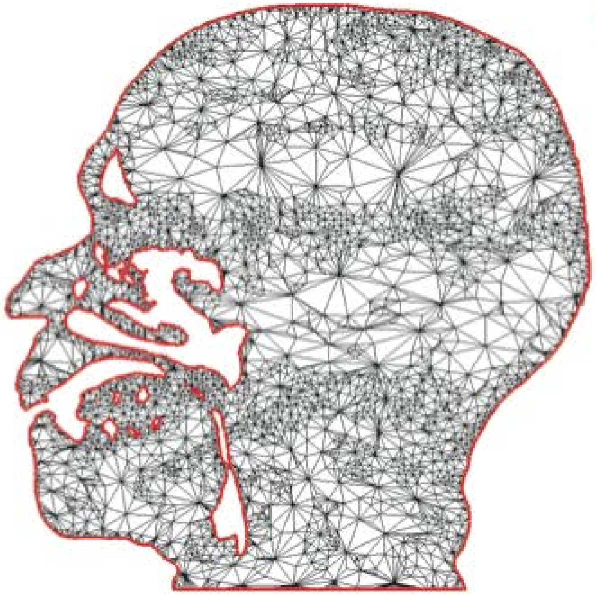


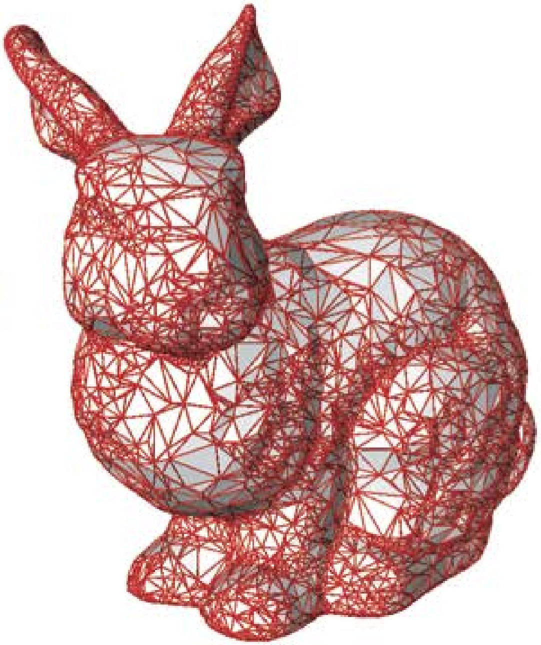
We cannot cover such a domain with a structured grid (why?)
•it's not of genus 0, so cannot be covered with a matrix-like distribution of cells

For this, we need unstructured grids (see next)

Unstructured grids







- different cell types can be mixed (though it's not usual)
- both vertex coordinates and cell themselves are freely (explicitly) specifiable
- implementation

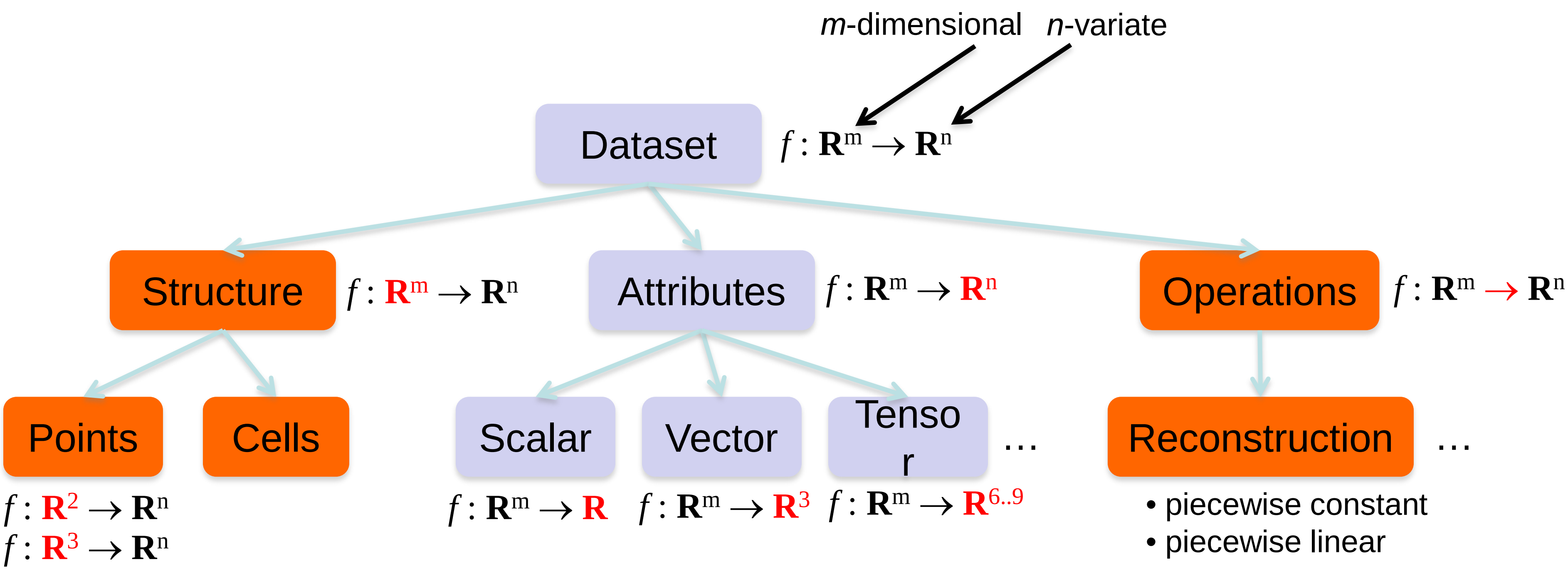
  vertex set $V = \{v_i\}$

  cell set $C = \{c_i = (indices\ of\ vertices\ in\ V)\}$
- most flexible, but most complex/expensive grid type

Storage requirements

$m\|V\| + s\|C\|$ for a m-dimensional grid with cells having s vertices each

Recapitulation: Dataset



- We discussed about **these** (grids, interpolation, reconstruction)
- We discuss next about **attributes**

Data attributes

$f : \mathbf{R}^m \rightarrow \mathbf{R}^n$

- n=0 no attributes (we model a shape only e.g. a surface)
- n=1 scalars (e.g. temperature, pressure, curvature, density)
- n=2 2D vectors
- n=3 3D vectors (e.g. velocity, gradients, normals, colors)
- n=6 symmetric tensors (e.g. diffusion, stress/strain – Modules 5..6)
- n=9 assymetric general tensors (not very common)

Remarks

- an attribute is usually specified for **all** sample points in a dataset (why?)
- different measurements will generate different attributes
- each attribute is interpolated separately
- different visualization methods for each n (see Module 3 next)

Data attributes: Color

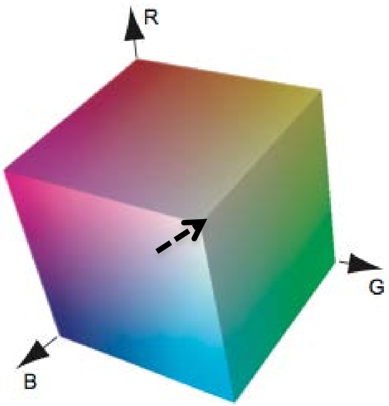
- complex topic (measurement, perception, representation)
- we'll mainly focus on representation and a bit on perception

RGB color system

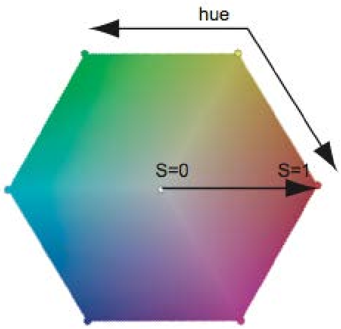
$$c = (c_R, c_G, c_B) \in [0,1]^3$$

- three floating-point components in $[0,1]$
- additive system (add, or mix, components to obtain result)

color cube



slice along plane
orthogonal
to main diagonal



color hexagon

- perfect for synthesis (e.g. in the graphics card)
- unintuitive for humans, who think easier in **hues**

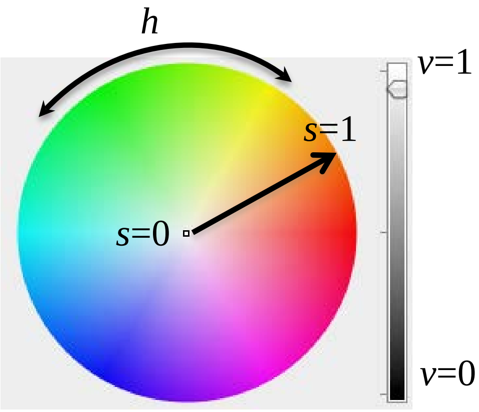
HSV color system

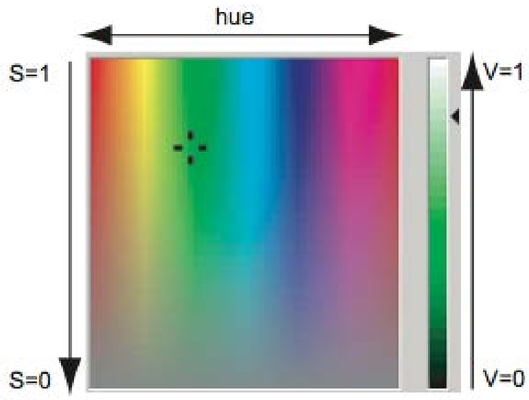
•three floating-point components in $[0,1]$

$$c = (h, s, v) \in [0,1]^3$$

•hue: tint of the color (red, green, blue, yellow, cyan, magenta, yellow, …)
•saturation: strong color (s=1), grayish color ($0<s<1$) or gray (s=0)
•value: luminance; white (v=1), dark ($0<v<1$), or black (v=0)

color wheel





color wheel stretched onto a square

•HSV widgets: typically specify h and s in a 2D canvas and v separately (slider)
•show a 'surface slice' in the RGB cube

RGB-HSV mapping

graphics software

RGB color

```
void rgb2hsv(float  r, float  g, float  b,
             float& h, float& s, float& v)
{
  float M = max(r,max(g,b));
  float m = min(r,min(g,b));
  float d = M-m;
  v = M;                    //value = max(r,g,b)
  s = (M>0.00001)? d/M:0;   //saturation
  if (s==0) h = 0;          //achromatic case, hue=0 by convention
  else                      //chromatic case
  {
    if (r==M)       h =     (g-b)/d;
    else if (g==M)  h = 2 + (b-r)/d;
    else            h = 4 + (r-g)/d;
    h /= 6;
    if (h<0) h += 1;
  }
}
```

Listing 3.2. Mapping colors from RGB to the HSV space.

```
void hsv2rgb(float  r, float  g, float  b,
             float& h, float& s, float& v)
{
  int   hueCase = (int)(h*6);
  float frac    = 6*h-hueCase;
  float lx      = v*(1 - s);
  float ly      = v*(1 - s*frac);
  float lz      = v*(1 - s*(1 - frac));
  switch (hueCase)
  {
    case 0:
    case 6: r=v;  g=lz; b=lx; break;  // 0<hue<1/6
    case 1: r=ly; g=v;  b=lx; break;  // 1/6<hue<2/6
    case 2: r=lx; g=v;  b=lz; break;  // 2/6<hue<3/6
    case 3: r=lx; g=ly; b=v;  break;  // 3/6<hue/4/6
    case 4: r=lz; g=lx; b=v;  break;  // 4/6<hue<5/6
    case 5: r=v;  g=lx; b=ly; break;  // 5/6<hue<1
  }
}
```

Listing 3.3. Mapping colors from HSV to the RGB space.

HSV color

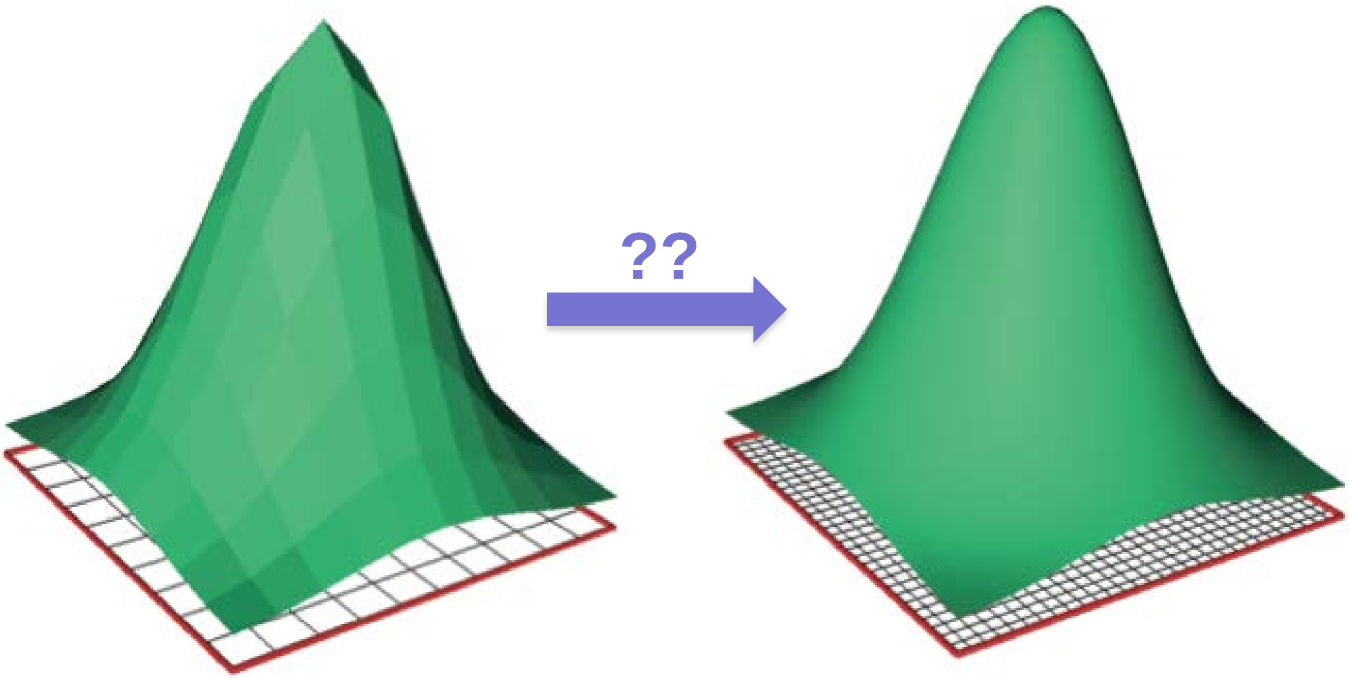
user

- simple conversions
- for details, see Chapter 3, pages 72-74

Advanced data representation issues

Data resampling

- consider building a Gouraud-shaded surface plot



Flat shading

- normals computed per **cell**

Gouraud shading

- normals required per **vertex**

- how to compute vertex attributes (normals) when we have cell attributes?

Data resampling: cell data to vertex data and back

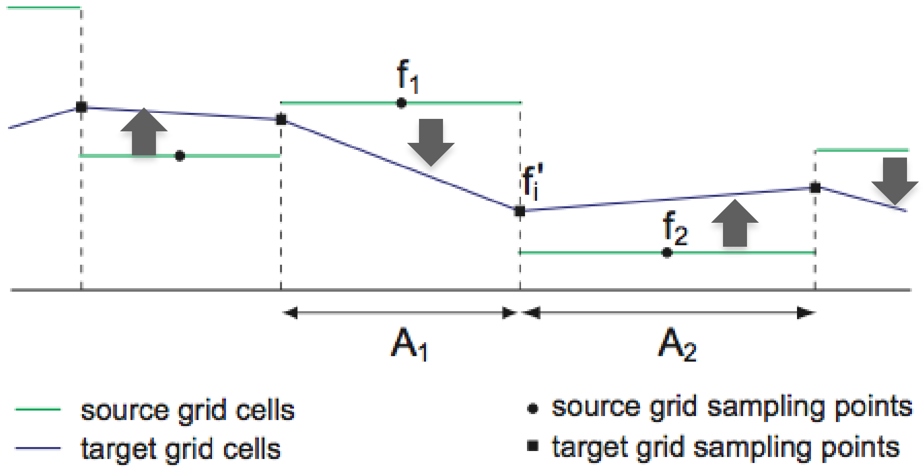


Figure 3.16. Converting cell to vertex attributes. The vertex value f_i' equals $\frac{A_1 f_1 + A_2 f_2}{A_1 + A_2}$, the area-weighted average of the cell values using vertex i.

Resampling a signal $\tilde{f}$ over some target domain D' should yield a 'similar' signal $\tilde{f}'$

$$\int_{c_i'} \tilde{f}' ds \approx \int_{c_i'} \tilde{f} ds, \quad \forall \text{ cells } c_i' \in \mathcal{D}' \xrightarrow[\text{see Sec. 3.9.1}]{} f_i' = \frac{\sum_{c_j \in \text{ cells}(p_i)} A(c_j) f_j}{\sum_{c_j \in \text{ cells}(p_i)} A(c_j)}$$

•this is the classical area-weighted normal averaging used in Gouraud shading

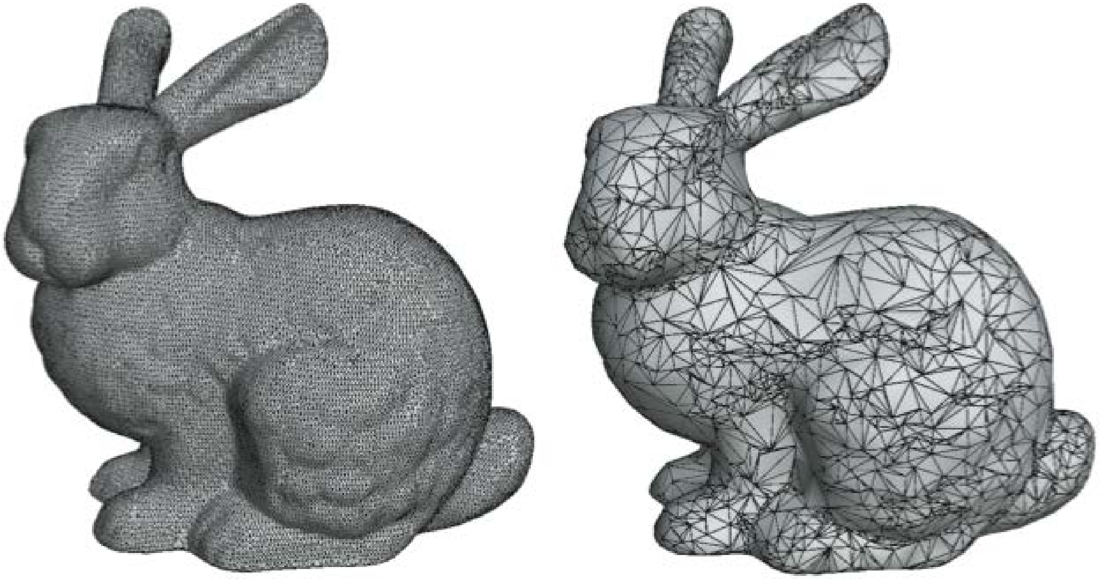
Resampling vertex data to cell data (same reasoning as above)

$$f_i = \frac{\sum_{p_j \in \text{ points}(c_i)} f_j'}{C}$$

•this is the classical averaging of vertex values to compute cell values

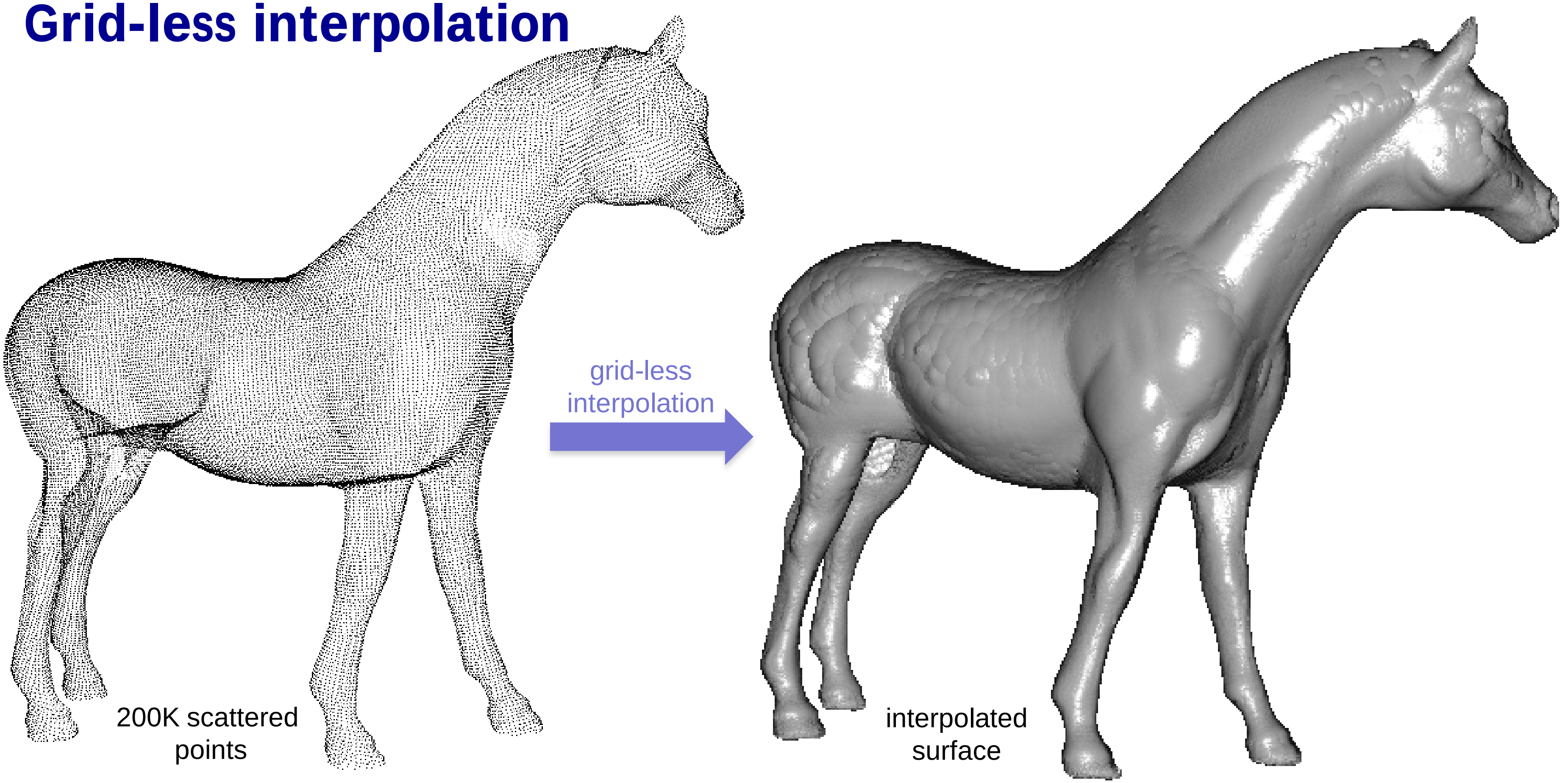
Data super/subsampling

- we have data on some grid
- we want data on a 'similar' grid having more or less cells
- the interpolation functions stay the **same** (unlike in resampling)



- this is an advanced topic – treated separately in Module 6

Grid-less interpolation



- we have data at some 'scattered' point locations in D
- we have no grid (cells connecting points)
- we can
  - construct such a grid (triangulation)
  - interpolate without a grid (radial basis functions)
- These advanced topics are discussed separately in Module 6

Summary

Data Representation (book Chapter 2)

- reconstruct continuous representations of sampled signals
    - efficiently
    - accurately

- interpolation, grids, and cells

- data attributes (scalars, vectors, tensors)

- advanced issues (resampling, grid-less interpolation)

- read Ch. 2 *in detail* to understand all the math!

Next module

- visualization algorithms

Happy so far?