Basic Techniques in Computer Graphics

Winter 2017 / 2018

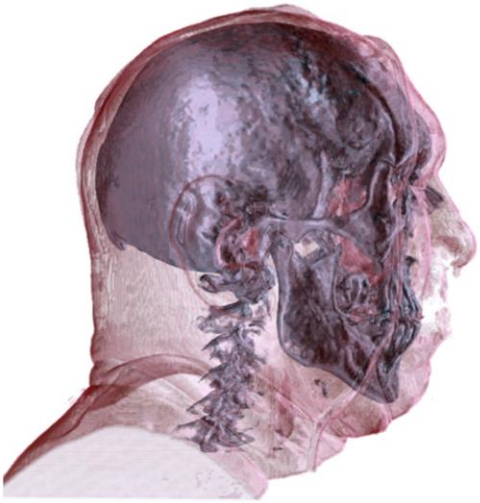
The slide comments are not guaranteed to be complete, they are no alternative to the lectures itself. So go to the lectures and write down your own comments!

Alternative Rendering Techniques

- **Volumetric Representations**
- Point-based representations

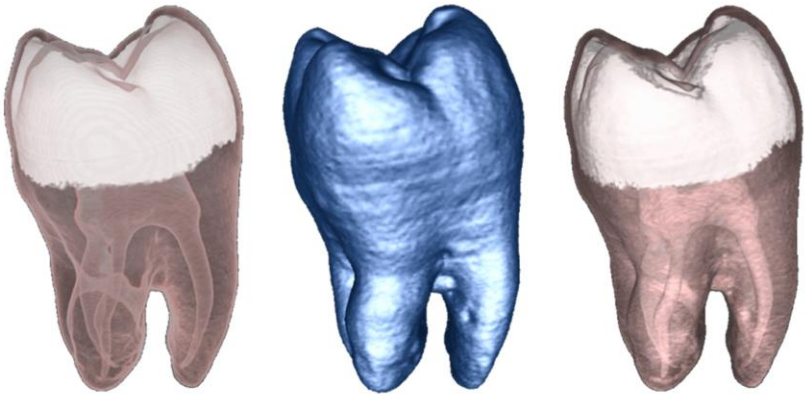
Not all objects can be expressed as a combination of primitives, in some domains a discrete volume has to be visualized.

Volume Rendering

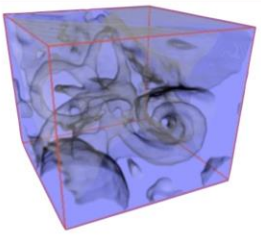


Medical imaging is one important domain of volume rendering.

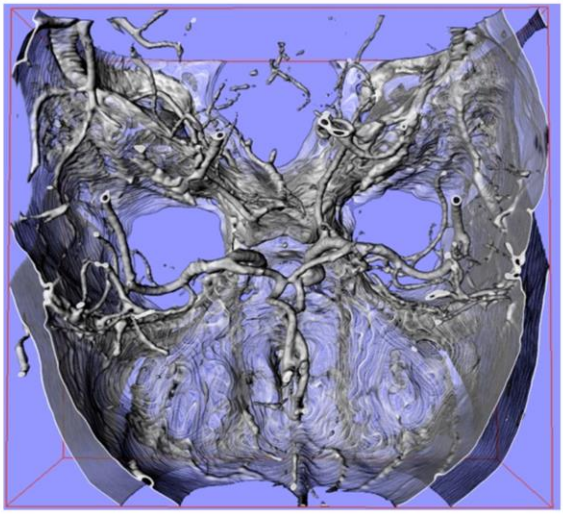
Volume Rendering

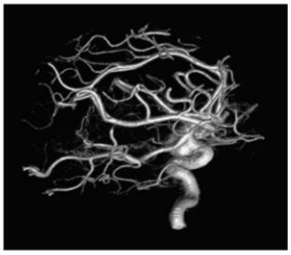


Volume Rendering



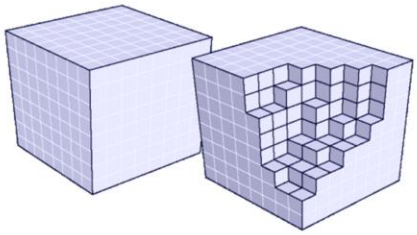
Volume Rendering



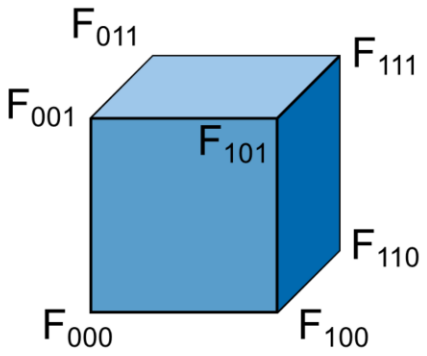


Volumetric Representations

- Implicit representation $F(x,y,z) = 0$
- Signed distance function
- Sample on a uniform cartesian grid
- Trilinear interpolation



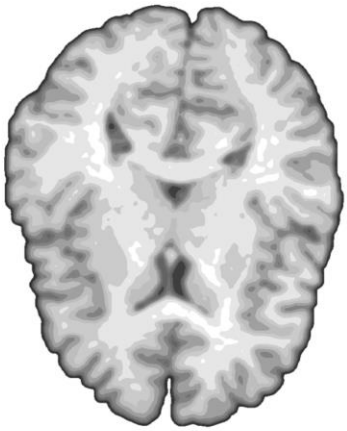
Volumetric Representations



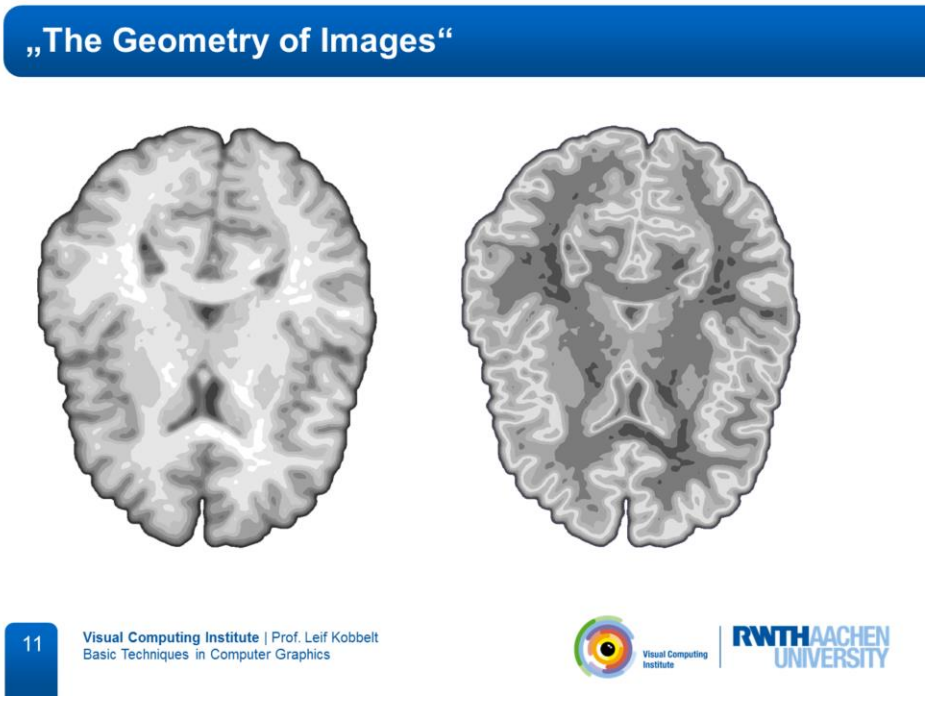
$$
\begin{aligned}
F(i+u,j+v,k+w) = \\
&F_{000}\,(1-u)(1-v)(1-w) + \\
&F_{100}\quad u\,(1-v)(1-w) + \\
&F_{010}\,(1-u)\quad v\,(1-w) + \\
&\quad\ldots \\
&F_{111}\quad u\quad v\quad w
\end{aligned}
$$

This is just the 3D extension of the bilinear interpolation we have already discussed for 2D textures.

„The Geometry of Images“

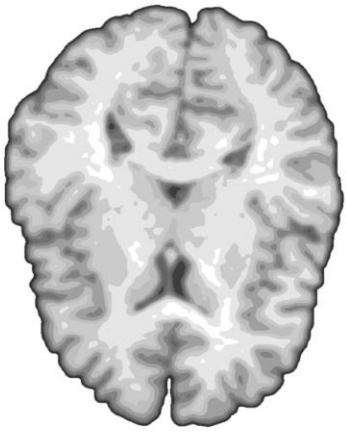


Saved in medical images from CT or MRT scans only contain density values of the tissue. The mapping to useful colors is up to the rendering process.



Right: A different mapping.

„The Geometry of Images“



Right: visualization of the contours.

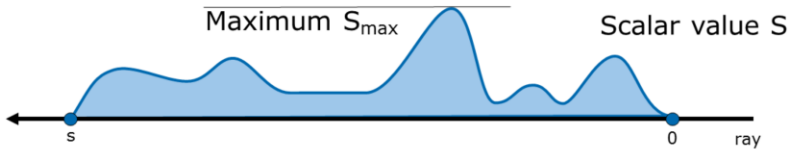
Volumetric Representations

- *Direct Rendering*:
  - Ray Casting } backward mapping
  - Slicing (3D Textures) } forward mapping
- *Indirect Rendering*:
  - Isosurface extraction: *Marching Cubes*

There are in general two ways of rendering volumes: directly take the data and render it or first extract a useful representation (like a triangle mesh of the objects with a certain density) and render that with the rendering pipeline.

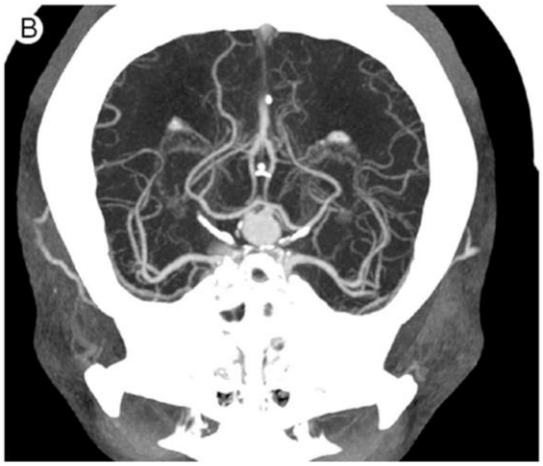
Ray Casting

- Maximum Intensity Projection



The intersection of the viewing ray and the volume is a function that should get mapped onto one color. One way would be to map the maximum value of the function to a gray value.

Maximum Intensity Projection

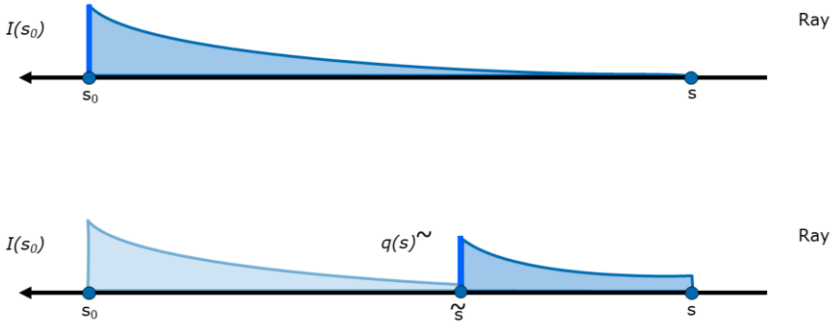


Here the maximum values are mapped to a color which map the bones to while as they have the highest density. Information about the distance of the maximum gets discarded in this mapping.

Absorption

Alternatively each voxel can be seen as one light source. The farther away the light, the stronger it should get damped as the ‚light' has to travel thru the volume.

Absorption



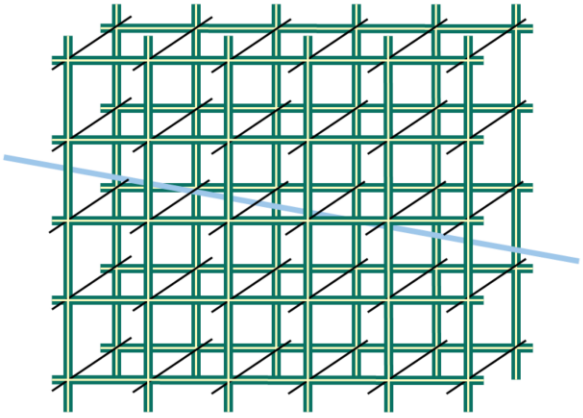
Shown here is the absorption of one value. In addition each cell should also get a opacity value so objects behind it will get additionally damped. For example bone tissue should be fully opaque and tissue behind it should not be visible.

Ray Casting

- *Image order*: for each pixel ...
  - Shoot rays from eye through each image plane pixel
  - Integrate color c & opacity μ along the ray r

$$c(r) = \int_0^L c(s) \cdot e^{-\int_0^s \mu(t)\,\mathrm{d}t}\,\mathrm{d}s$$

Volumetric Representations



As my volume data is discrete anyway the integrals can get replaced by sums.

Opacity

- base color c, opacity μ

  → damping factor $1-\alpha = e^{-\mu}$
  → emitted color $\alpha\, c$

- resulting (effective) color

  → $\alpha\, c + (1-\alpha)\, c_{background}$

Opacity Accumulation

- **Continous** compositing

$$c(u_i) = \int_0^L \alpha \cdot c(\mathbf{E} + \lambda \mathbf{D}) \cdot e^{-\int_0^\lambda \mu(\mathbf{E}+t\mathbf{D})\,\mathrm{d}t}\,\mathrm{d}\lambda$$

- **Discrete** compositing

$$c(u_i) = \sum_{j=0}^{K} \left[\alpha(x_j) \cdot c(x_j) \cdot \prod_{m=0}^{j-1} (1 - \alpha(x_m)) \right],$$

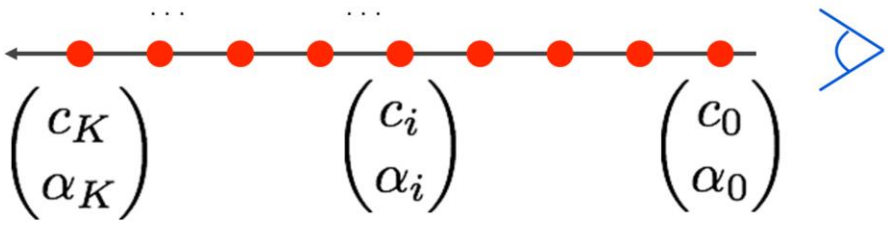
$$\text{with } c(x_K) = c_{\text{bkg}}, \quad \alpha(x_K) = 1$$

Ray Casting

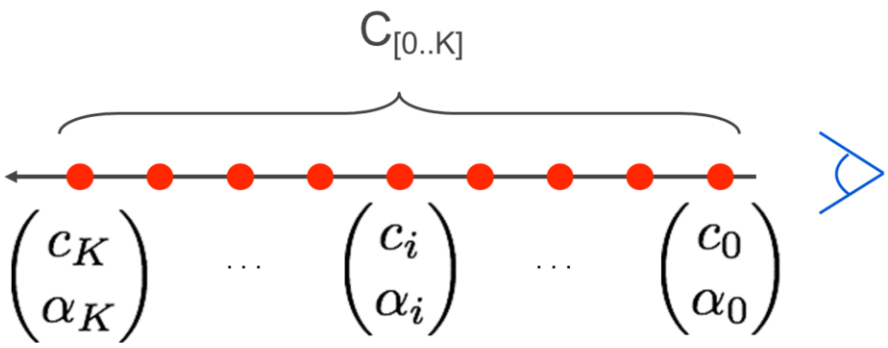
set color $c(.)$ and opacity $\alpha(.)$ for all voxels

for all pixels $\boldsymbol{u}_i = (u_i, v_i)$
  find ray from eye through pixel $\boldsymbol{u}_i$: eye + $\lambda \boldsymbol{d}$
  init $c_{acc}(\boldsymbol{u}_i) = \alpha_{acc}(\boldsymbol{u}_i) = 0$
  for all samples $\boldsymbol{x}_j = (x_j, y_j, z_j) = \text{eye} + \lambda_j \boldsymbol{d}$
    get $c(\boldsymbol{x}_j)$ and $\alpha(\boldsymbol{x}_j)$ by tri-linear interpolation
    composite with $c_{acc}(\boldsymbol{u}_i)$ and $\alpha_{acc}(\boldsymbol{u}_i)$

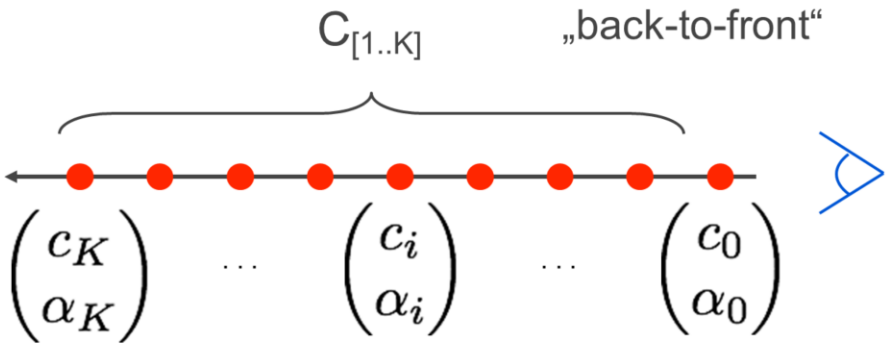
Opacity Accumulation



Opacity Accumulation



Opacity Accumulation



Front-to-back accumulation can be more efficient as the accumulation can be stopped as soon as a full opaque object got hit (or a mostly opaque if a small error is ok).

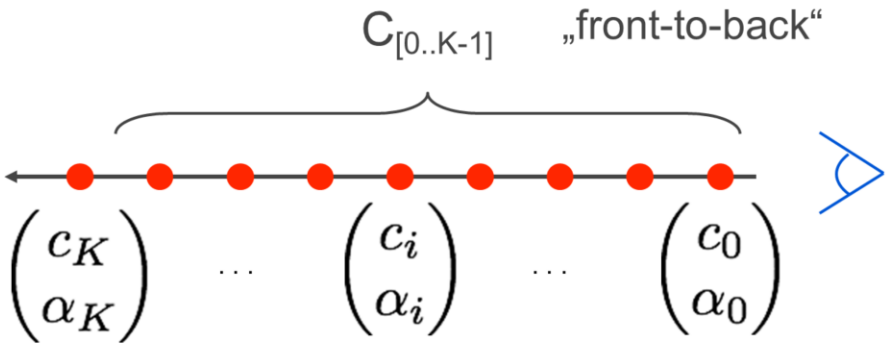
Back-to-Front Compositing

$$\sum_{j=0}^{K} \left[c(x_j) \cdot \alpha(x_j) \cdot \prod_{m=0}^{j-1} (1 - \alpha(x_m)) \right]$$

$$= c(x_0) \cdot \alpha(x_0) +$$

$$\sum_{j=1}^{K} \left[c(x_j) \cdot \alpha(x_j) \cdot \prod_{m=0}^{j-1} (1 - \alpha(x_m)) \right] \cdot (1 - \alpha(x_0))$$

Opacity Accumulation



Front-to-Back Compositing

- Iterative accumulation j = 0 .. K

$$
\begin{aligned}
C &\leftarrow C + \alpha_j \cdot (1 - \hat{\alpha}) \cdot c_j \\
\hat{\alpha} &\leftarrow \hat{\alpha} + \alpha_j \cdot (1 - \hat{\alpha})
\end{aligned}
$$

with

$$
(1 - \hat{\alpha}) = \prod_{m=0}^{j-1} (1 - \alpha_m)
$$

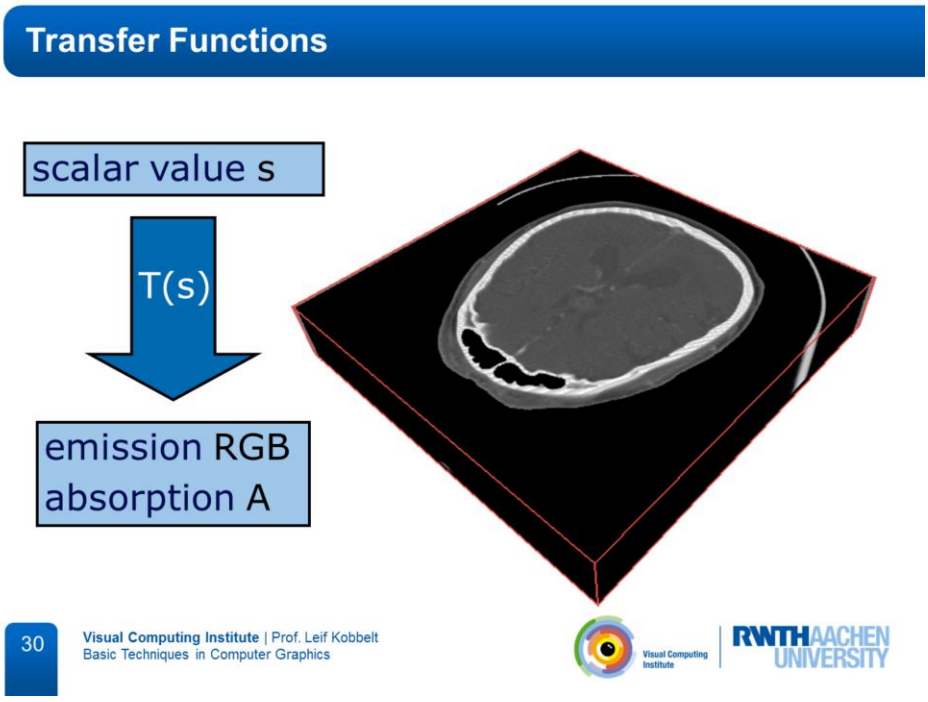
Transfer Functions

- how to define the
  - base color **c**
  - opacity α
- for each voxel ?

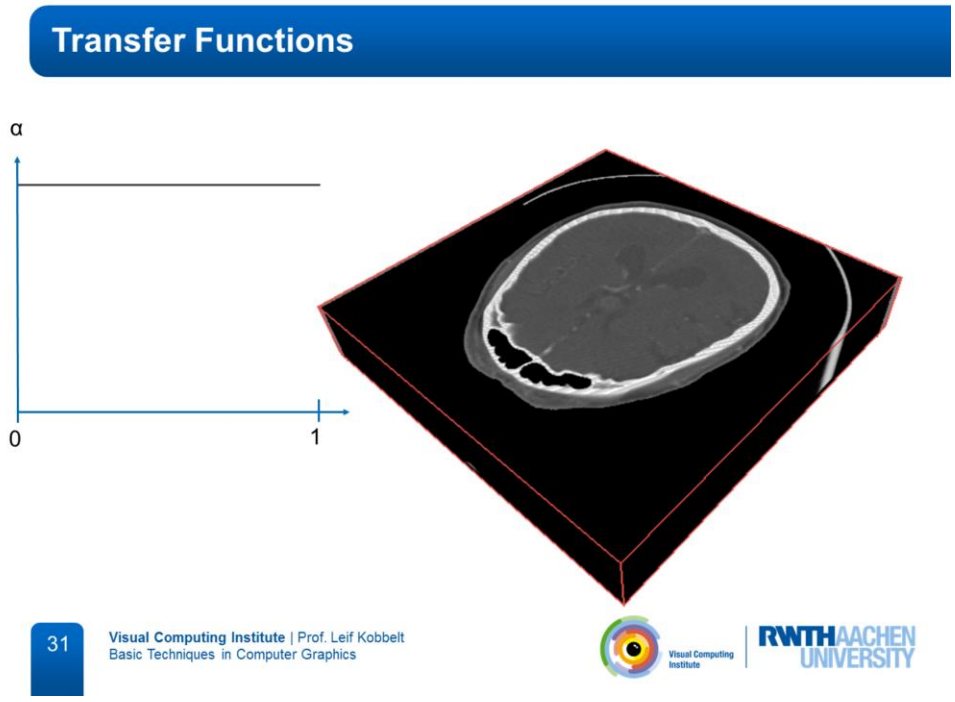
- CT, MRT, ...

- diffusion tensor imaging

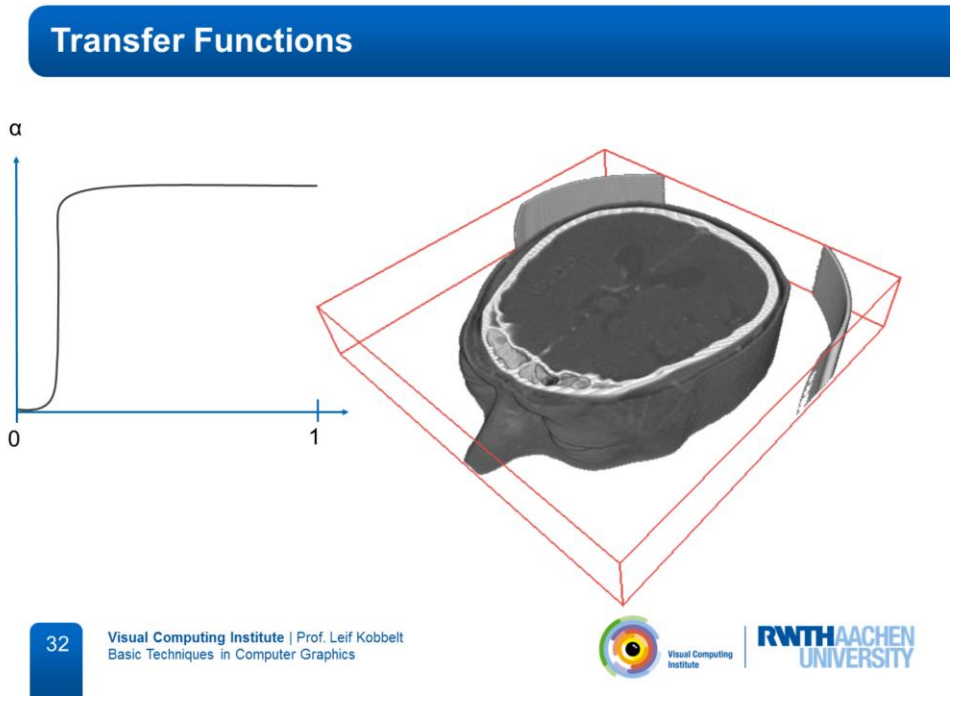
The mapping from the density values that the (CT/MRT) scan created to a color or opacity value is called the transfer function.



Given: scalar values s
Transfer function T(s) maps these values to an emitted RGB value and absorption value A.

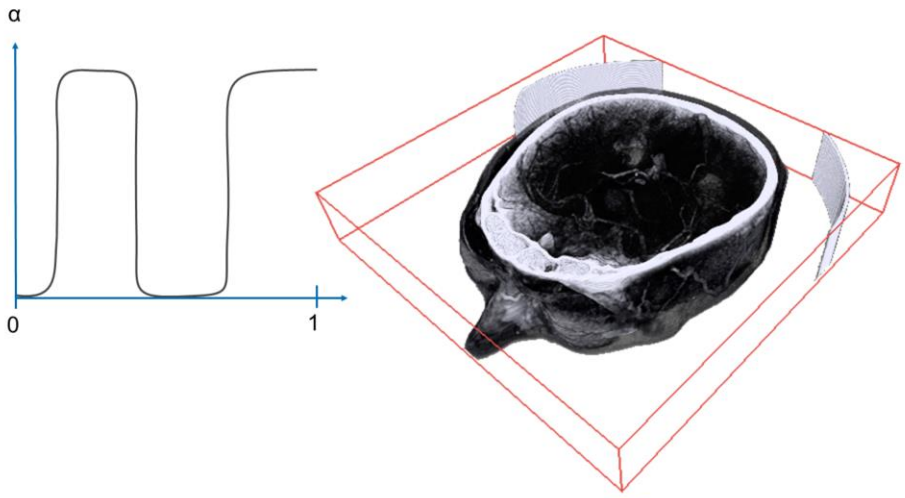


Full opacity at all voxels.
The bone is white, brain and skin is grey, air is black (no density).



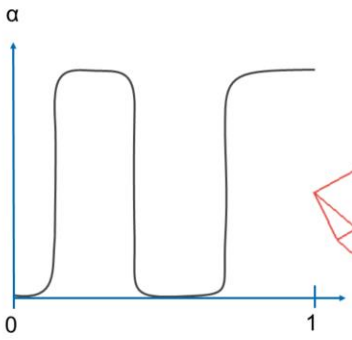
opacity alpha for surrounding air -> 0
Now more of the outer surface is visible.

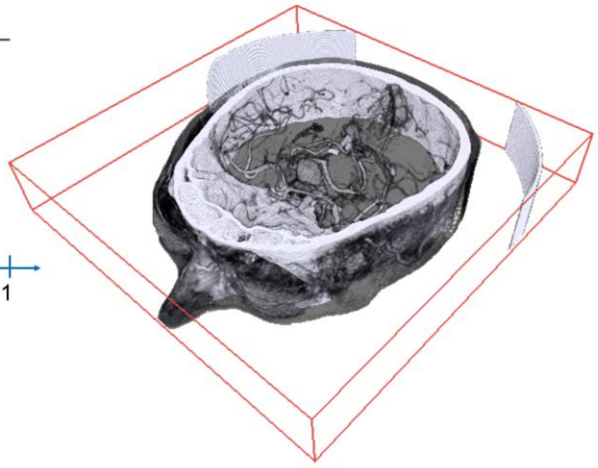
Transfer Functions



opacity alpha for gray and white brain tissue -> 0
This makes the blood vessels more visible.

Transfer Functions

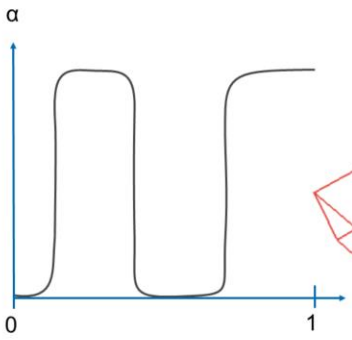


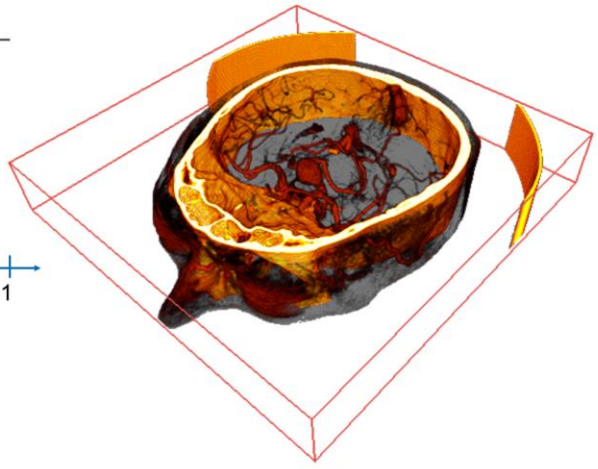


+ Shading

Adding shading and a fake light source.

Transfer Functions

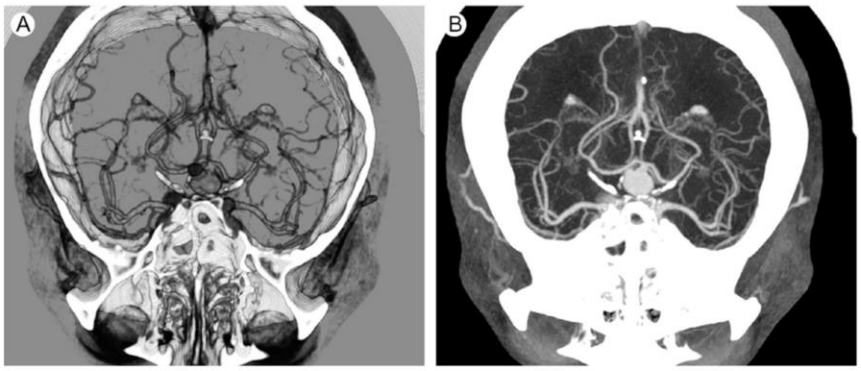




+ Shading
+ Color coding

Color Coding : Map density values to colors

Absorption vs. MIP



Left: Shaded rendering.
Right: Maximum intensity projection.

Volume Shading

- Lighting / Shading
  - Phong model
  - Estimate normal by $N(x,y,z) = \dfrac{\nabla F(x,y,z)}{\|\nabla F(x,y,z)\|}$
  - Approximate $\nabla F(x,y,z)$ by *central difference*

$$\nabla F(x_i,y_i,z_i) = \frac{1}{2}\begin{pmatrix} F(x_{i+1},y_i,z_i) & - & F(x_{i-1},y_i,z_i) \\ F(x_i,y_{i+1},z_i) & - & F(x_i,y_{i-1},z_i) \\ F(x_i,y_i,z_{i+1}) & - & F(x_i,y_i,z_{i-1}) \end{pmatrix}$$

Ray Casting

Acceleration techniques

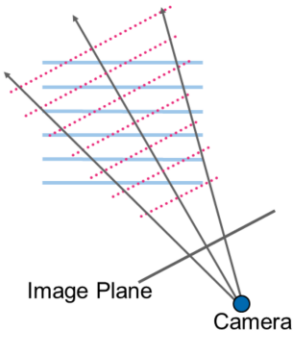
- Spatial data structures (octree)
  - Find and skip empty regions
  - Find homogenous regions, use lower sample rate
- *Early ray termination*
  - Opacity threshold terminates ray traversal
  - Requires front-to-back traversal
- Fast cell traversal (Bresenham-like)

Volumetric Representations

- *Direct Rendering*:
  - Ray Casting } backward mapping
  - Slicing (3D Textures) } forward mapping
- *Indirect Rendering*:
  - Isosurface extraction: *Marching Cubes*

Slicing

- Use volume dataset as 3D texture
- Draw slices
  - parallel to image plane
  - back-to-front order
  - accumulate in frame buffer
- 3D graphics hardware

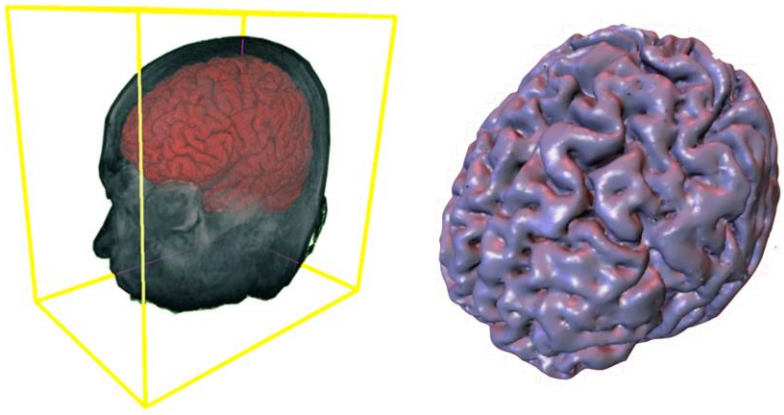


With 3D textures the ray casting can get approximated with older graphics hardware (on modern hardware the ray casting can get also implemented inside of a fragment shader directly). Each slice represent one set of samples with one sample per ray.

Volumetric Representations

- *Direct Rendering*:
  - Ray Casting } backward mapping
  - Slicing (3D Textures) } forward mapping
- *Indirect Rendering*:
  - Isosurface extraction: *Marching Cubes*

Direct vs Indirect Rendering



An example of a volume (left) and the extracted isosurface on the right.

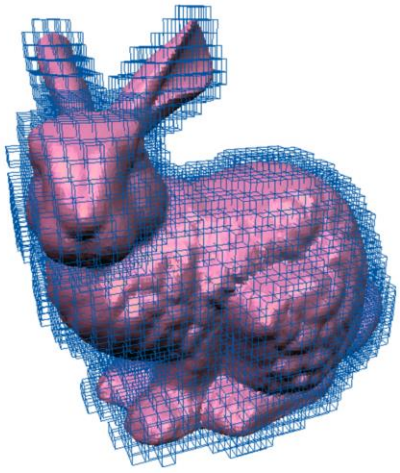
Indirect Volume Rendering

- Extract iso-surface from volume
- Represented as triangle mesh
  - surface sample positions
  - mesh connectivity
- Use graphics hardware for rendering
- Generate patch for each non-empty cube

  → *Marching Cubes*

Visit Each Gray Cell ...

- Cell classification:
  - inside (white)
  - outside (black)
  - intersecting (gray)

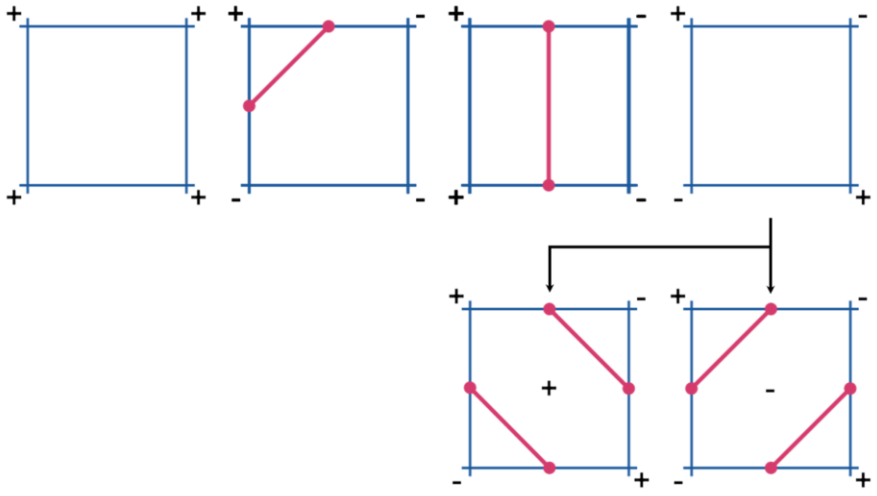


Marching *Squares*

- Classify cell corners as inside / outside
- Classify cell: ***16*** configurations
- Linear interpolation along the edges
- Look-up table for edge configuration

Let‘s first look at the simpler 2D case called marching squares.

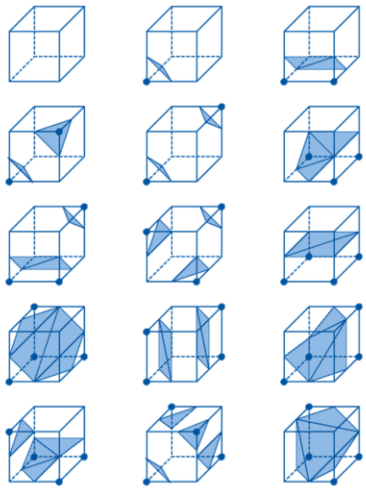
Marching *Squares*



Some extra cases are just rotations or flipped signs. For the rightmost case two connections of the intersections can be found.

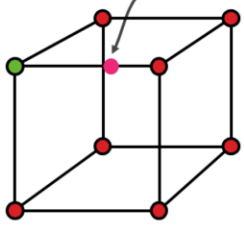
Marching Cubes

- ***256*** configurations
- Look-up table:
  2^8 entries
- Disambiguation is more complicated



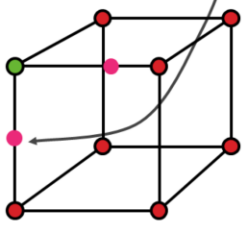
... Compute Samples on Edges ...

$$\left(i + \frac{|d_{i,j,k}|}{|d_{i,j,k}| + |d_{i+1,j,k}|}, j\ k \right)$$



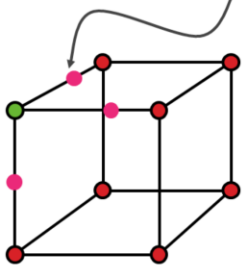
... Compute Samples on Edges ...

$$\left(i,\ j + \frac{|d_{i,j,k}|}{|d_{i,j,k}| + |d_{i,j+1,k}|},\ k\right)$$

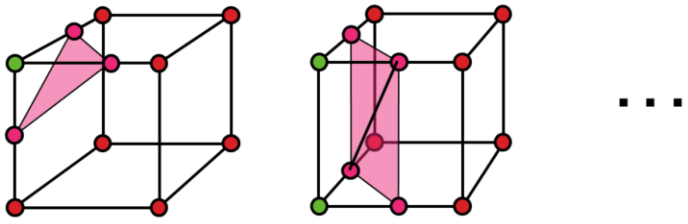


... Compute Samples on Edges ...

$$\left(i,\ j,\ k + \frac{|d_{i,j,k}|}{|d_{i,j,k}| + |d_{i,j,k+1}|}\right)$$



... look up mesh connectivity



Marching Cubes

- Resulting surface
  - Up to 5 triangles per cell
  - Uniform (over-) sampling
  - Badly shaped triangles (needles)
- Post-processing
  - Mesh decimation
  - Mesh optimization

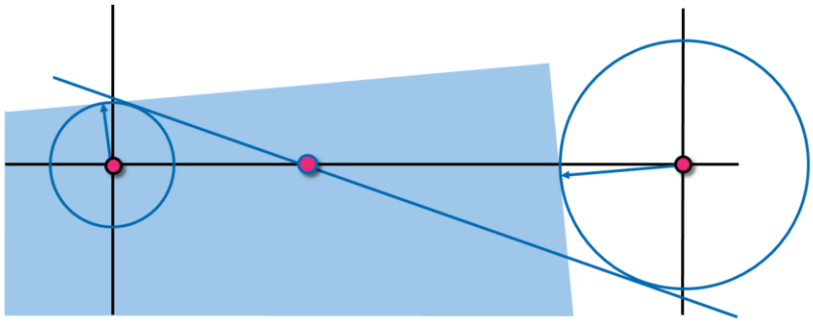
Shortcomings

- Scalar distance values at the grid points provide **insufficient approximation near sharp features**

- **Alias effects** due to global alignment of the samples to the grid cell edges

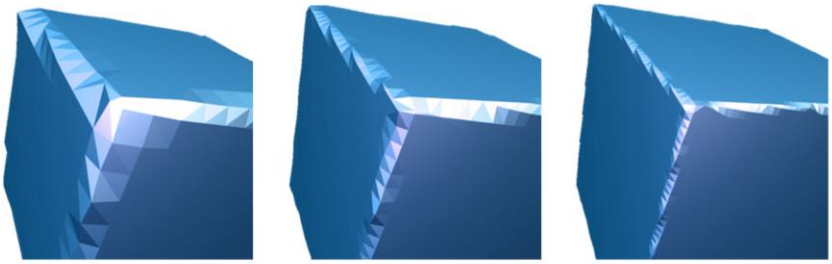
Shortcomings

Linear Interpolation

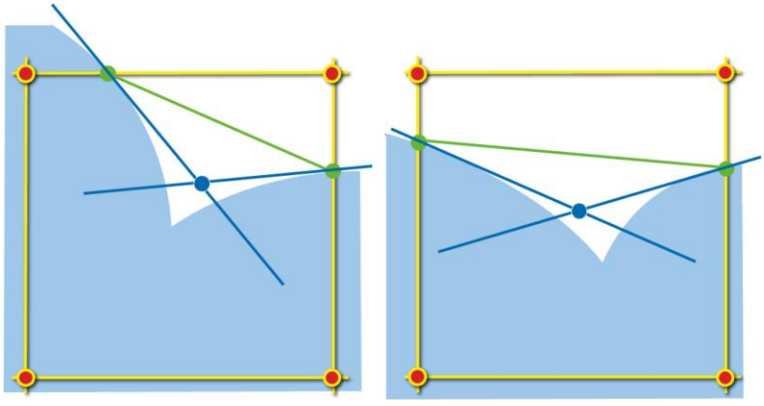


Shortcomings

Alias Errors



Feature Sampling



Extended Marching Cubes

For each cell ...

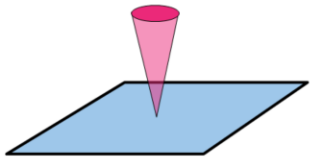
1. Find surface samples on the grid cell edges
2. Evaluate surface normals at the sample points
3. Feature ***detection***
4. Feature ***sampling***
5. Feature ***reconstruction***

Computing Surface Normals

- For the application scenarios
  - Constructive solid geometry
  - Point clouds
  - Polygonal meshes
- normal vectors can be found by evaluating the gradient of the distance function

- For each grid cell we get a set of edge samples $\{ p_i \}$ and normals $\{ n_i \}$

Feature Classification

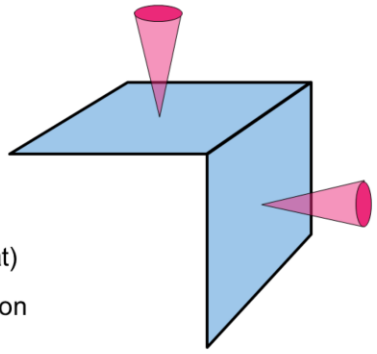


Normal cone

for a smooth (flat)

non-feature region

Feature Classification

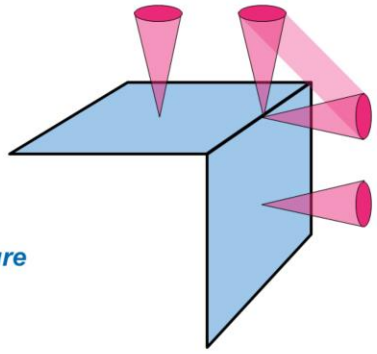


Normal cone

for a smooth (flat)

non-feature region

Feature Classification

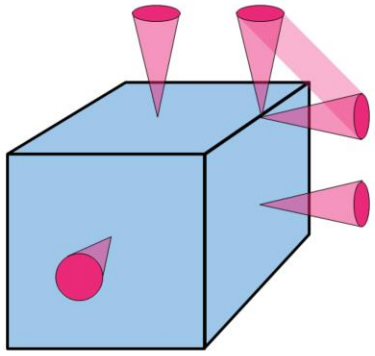


Normal cone

for an ***edge feature***

(***one*** large angle)

Feature Classification

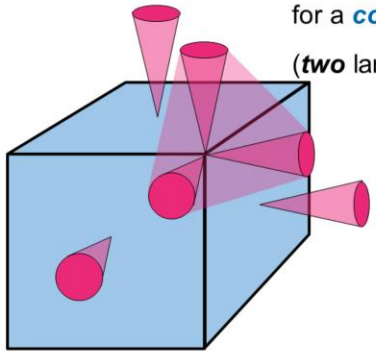


Feature Classification

Normal cone

for a ***corner feature***

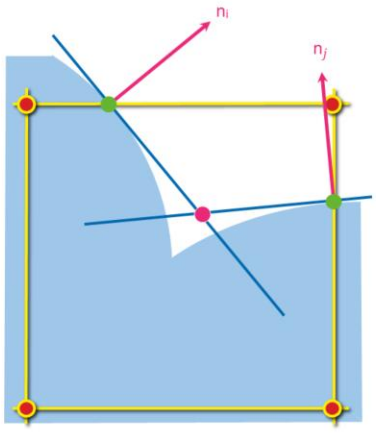
(***two*** large angles)



Feature Detection

- Opening angles of the normal cone
  $\theta = \min_{i,j} \langle n_i, n_j \rangle$
  $\varphi = \max_{k} \langle n_k, n_i \times n_j \rangle$
- Edge features
  $\theta < \theta_0, \ \varphi < \varphi_0$ → sharp *edge*
- Corner features
  $\theta < \theta_0, \ \varphi \geq \varphi_0$ → sharp *corner*

Feature Sampling



Feature Sampling

- Each pair (p_i, n_i) defines a ***tangent element***
- Intersect tangent planes to approximate a ***piecewise*** smooth surface
- Find least squares solution by **SVD**
  - Overdetermined cases
  - Underdetermined cases
  - Suppress smallest singular value at edge features

Singular Value Decomposition

- $M \in R^{m \times n}$

  $\rightarrow M = U \Sigma V^T$

- $U \in R^{m \times n}$ $\Sigma \in R^{n \times n}$ $V \in R^{n \times n}$
- $U^T U = Id_n$ $V^T V = Id_n$
- $\Sigma = diag(\sigma_1, \sigma_2, \sigma_3, ...)$

Singular Value Decomposition

- $M \in R^{m \times n}$

  → $M = U \Sigma V^T$

- $U \in R^{m \times m}$ $\Sigma \in R^{m \times n}$ $V \in R^{n \times n}$

- $U^T U = Id_m$ $V^T V = Id_n$ "rotation matrices"

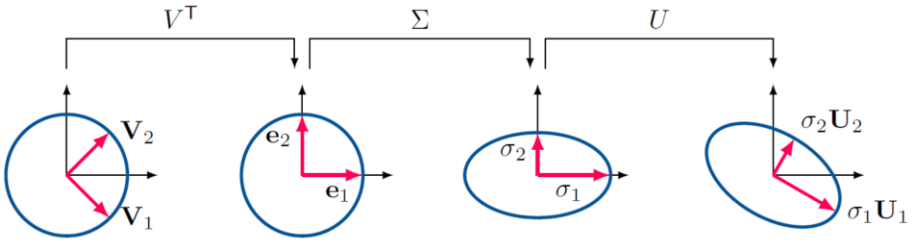
- $\Sigma = diag(\sigma_1, \sigma_2, \sigma_3, ...)$

Singular Value Decomposition

- $M \in R^{m \times n}$

  → $M = U \Sigma V^T$

- $U \in R^{m \times m}$ $\Sigma \in R^{m \times n}$ $V \in R^{n \times n}$



Pseudo Inverse

- $M \in R^{m \times n}$

  → $M = U \Sigma V^T$

- $M^{-1} = V \Sigma^* U^T$

- $\Sigma^* = \text{diag}(1/\sigma_1, 1/\sigma_2, 1/\sigma_3, ...)$

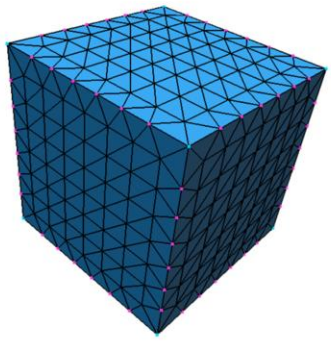
- $1/\sigma_i = 0$ if $\sigma_i < \varepsilon$

Least Squares / Least Norm

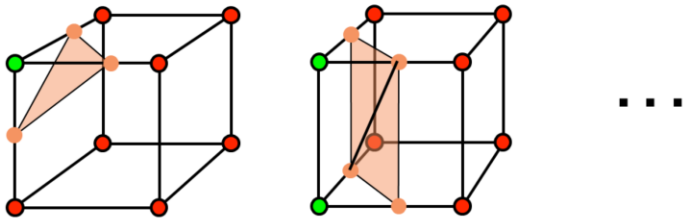
- $M\,x = U\,\Sigma\,V^T\,x = b$
- $x = M^{-1}\,b = V\,\Sigma^*\,U^T\,b$
- regular ($m = n$)
- overdetermined ($m > n$)
- underdetermined ($m < n$ or $\xi_i = 0$)

Feature Sampling

- approximately intersect tangent planes
  - underdetermined
  - regular
  - overdetermined

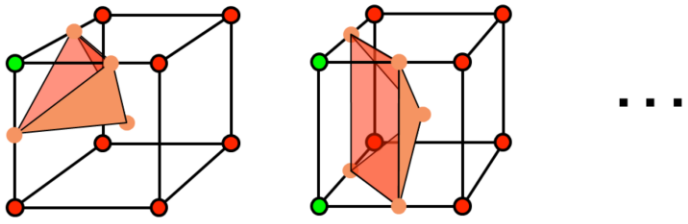


... Look-up Mesh Connectivity



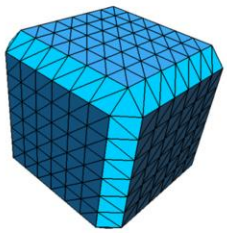
Modified Look-up Table

- Generate triangle fans centered around the feature sample

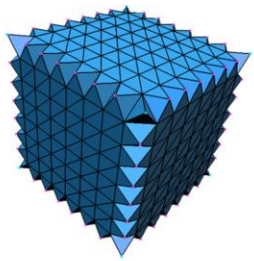


Feature Reconstruction

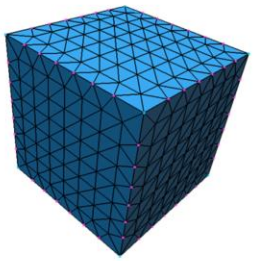
feature detection



feature sampling



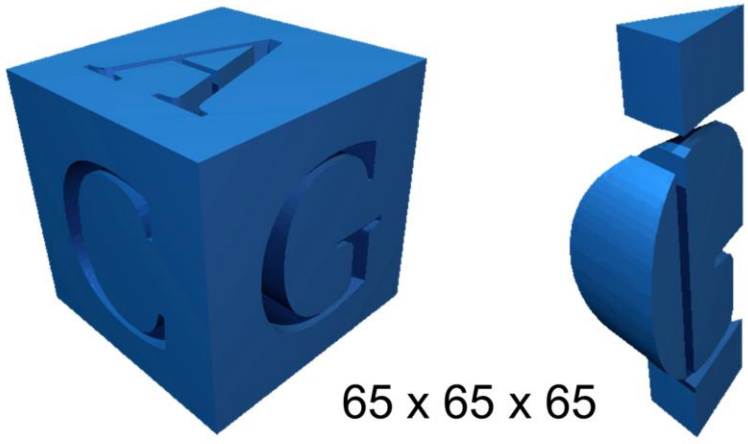
edge-flipping



Extended Marching Cubes

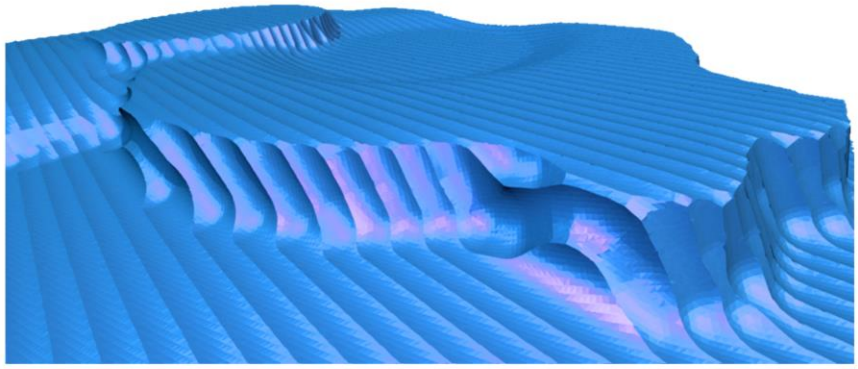
- Mesh complexity
  - Two additional triangles per feature sample
  - Typically 10% overhead
- Computation time
  - 20% to 40% overhead
- Improved approximation
- Parameters θ_0 and φ_0
  - False positive feature detection !?

CSG ... CAD / CAM



CSG ... Milling Simulation

257 x 257 x 257



Alternative Rendering Techniques

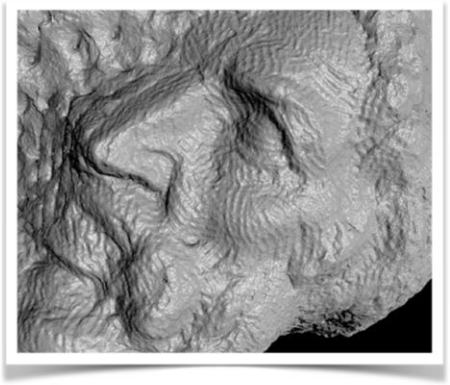
- Volumetric Representations
- **Point-based representations**

Point-Based Rendering

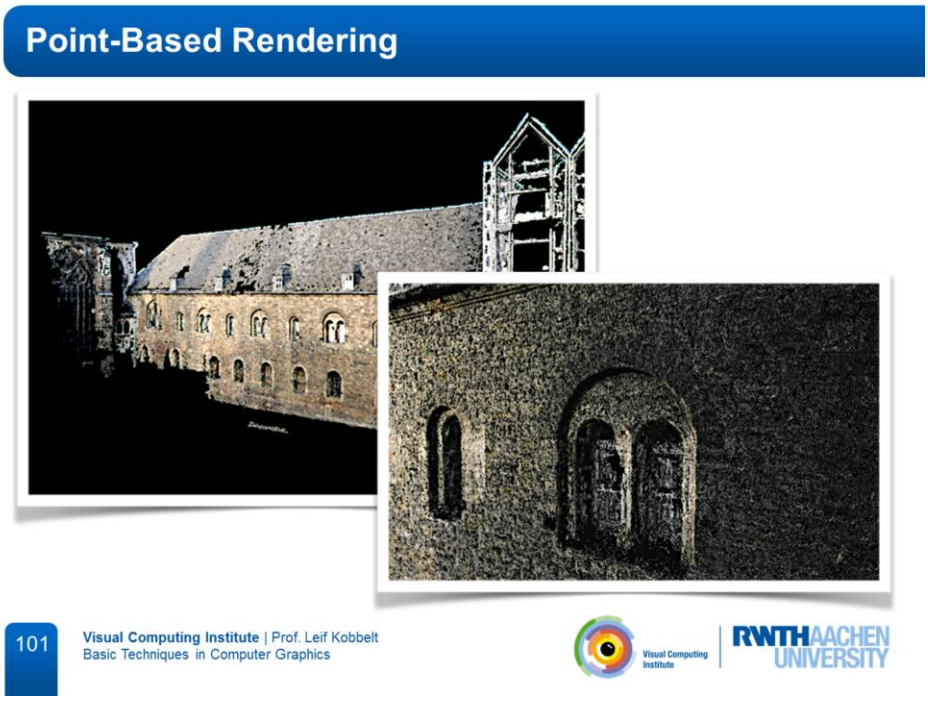
- number of pixels on screen: 2M (HD) – 8M (4k)
- number of triangles ...
- if each triangle covers less than a pixel
  - overhead by storing connectivity
  - overhead by scan conversion setup
- render just points instead ?!

#pixels on screen: ~2M at FullHD, often less

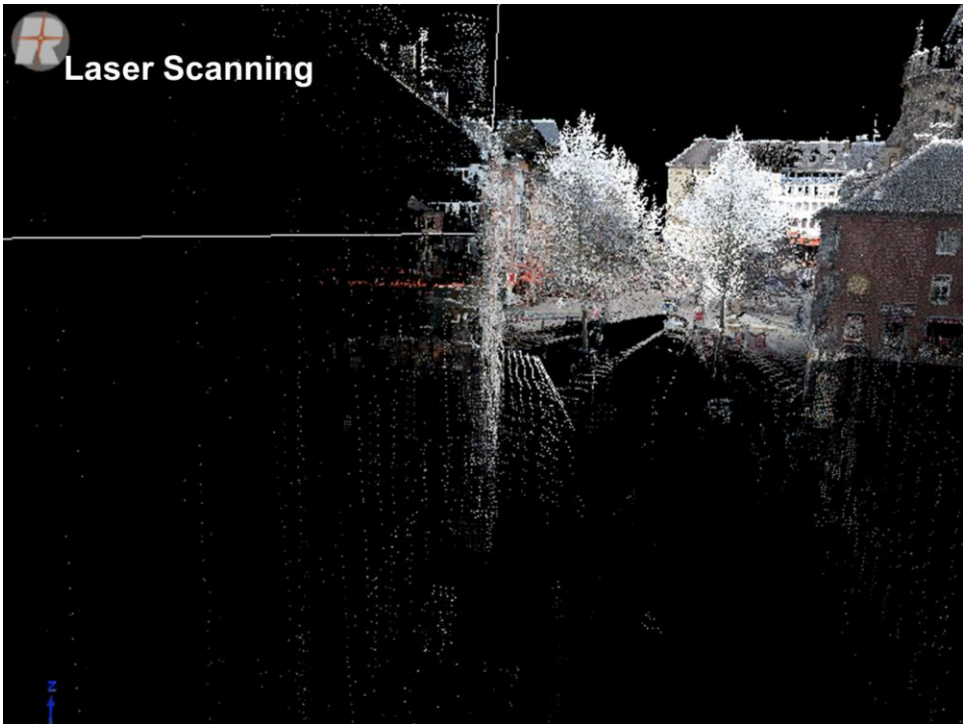
Point-Based Rendering

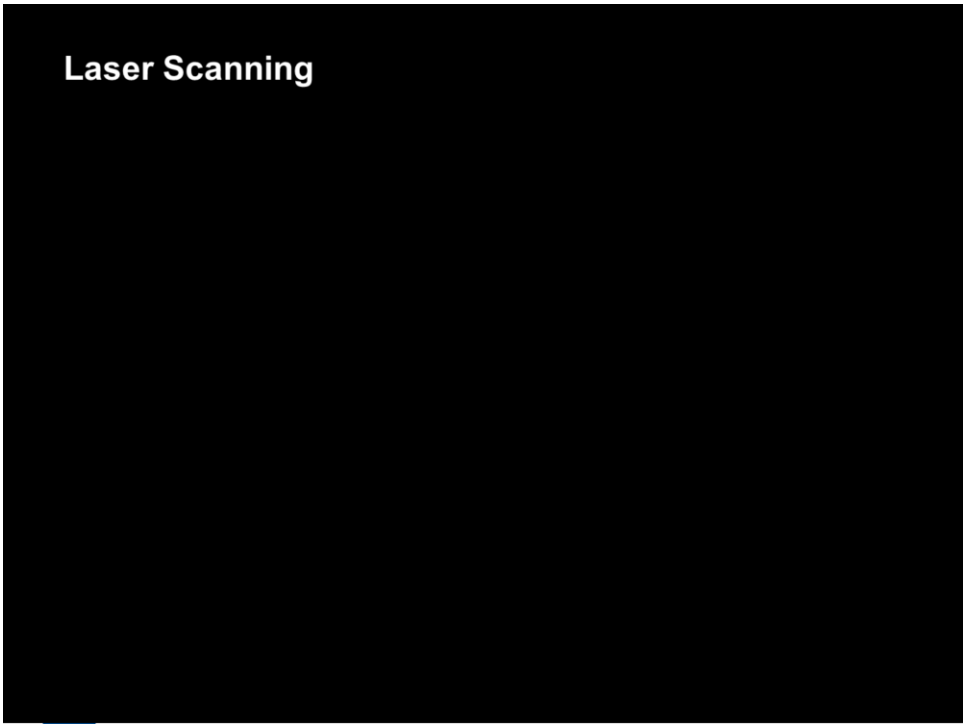


~400.000.000 triangles rendered onto a screen of max. 2M pixels, most likely less...



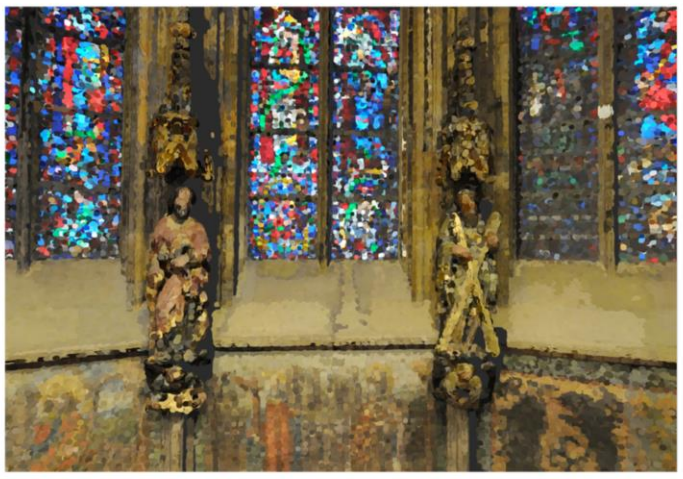
Naive rendered points will show holes at close ups. Filling these is a mayor topic of point based rendering.
Statistically there are enough points to cover all pixels, but they are not evenly distributed (and they can‘t be for all possible camera positions!)


Laser Scanning

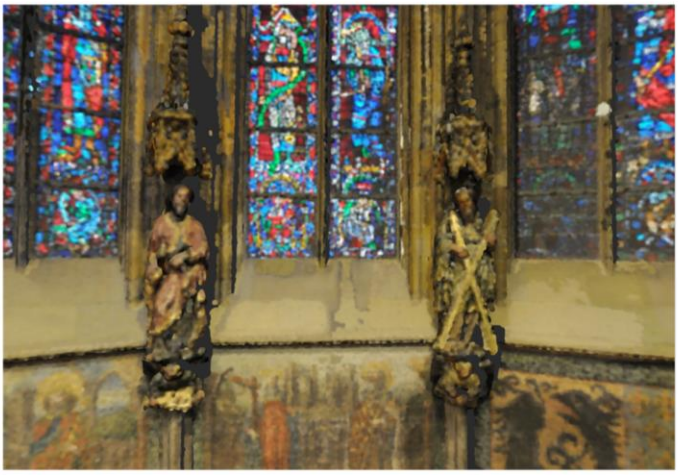

Laser Scanning

Point vs. Splat Rendering

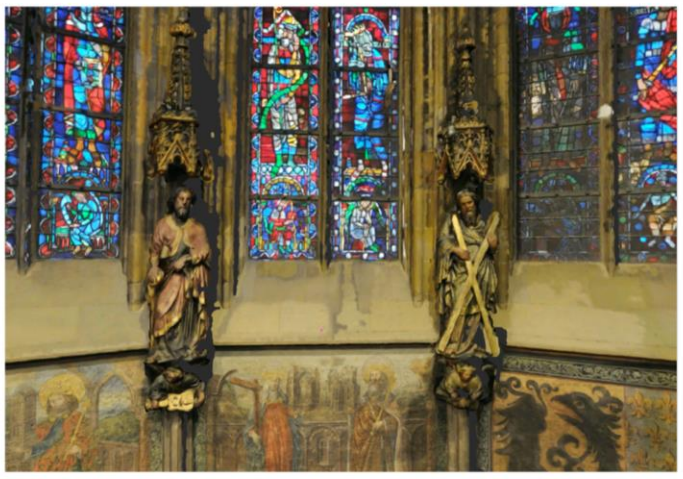
Point vs. Splat Rendering



Point vs. Splat Rendering

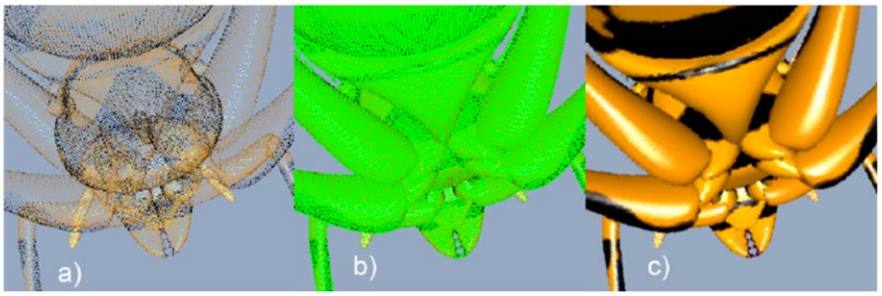


Point vs. Splat Rendering



Screen-Space Hole Filling

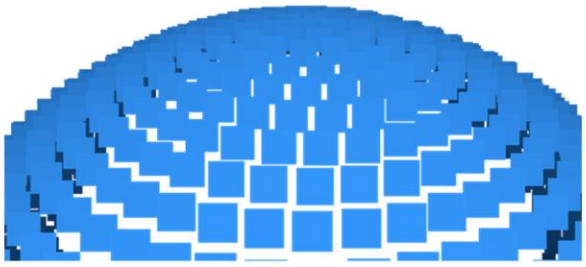
Points are not distributed uniformly!



Render ***splats*** to fill the gaps ...

An example of point rendering:
a) the points rendered as one pixel large points
b) the space highlighted in green is the area that should get filled if a closed surface should get rendered
c) the points rendered as splats to fill these holes.

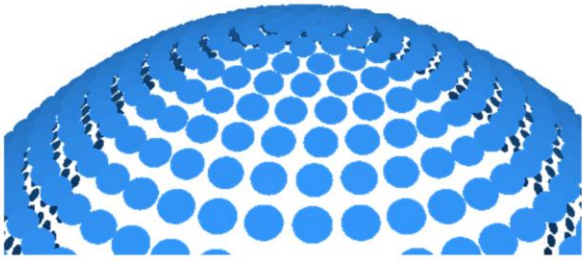
Splat Rendering



Screen-space vs. Object-space ...

splats in screen-space: all quads are oriented towards the camera. Of course circles in screen-space are also possible, here they are rendered as quads because OpenGL supports point-rendering with large points but draws them as quads...

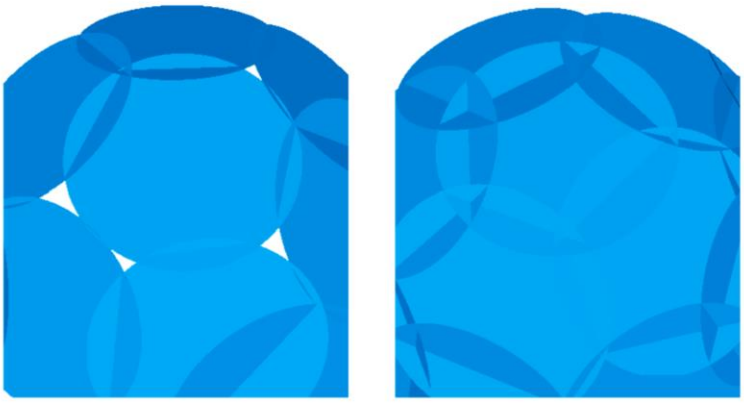
Splat Rendering



Screen-space vs. **Object-space** ...

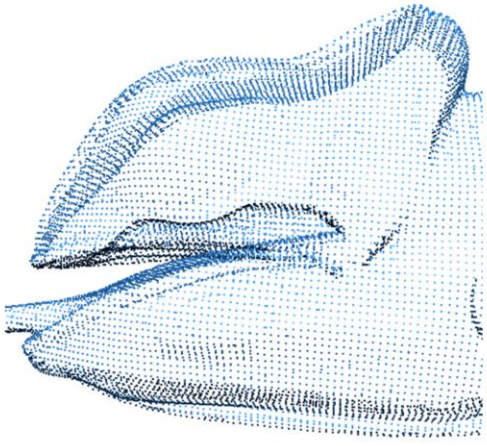
splats in object-space: the discs are also oriented to a normal that is stored for each point

Splat Rendering



The discs have to be large enough to fill all holes.

Splat Rendering

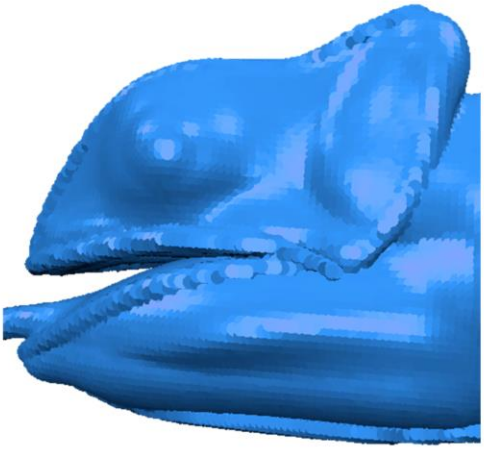


The points rendered as one pixel large points

Splat Rendering

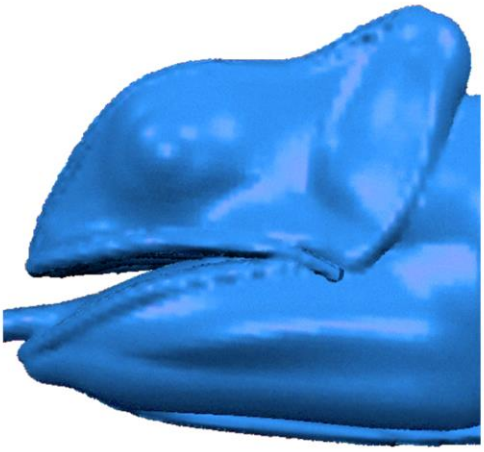
screen space quads

Splat Rendering



Object-space disks

Splat Rendering

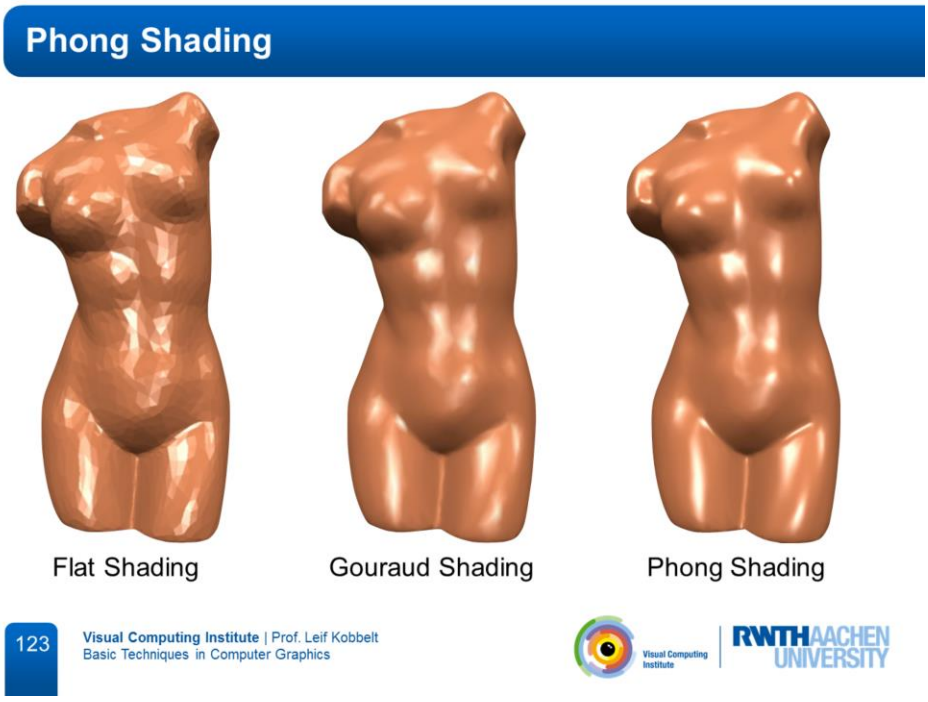


Filtering the image to make the surface more smooth. Note that the silhouette did not change in this step while it changed between the screen-space and object-space rendering.

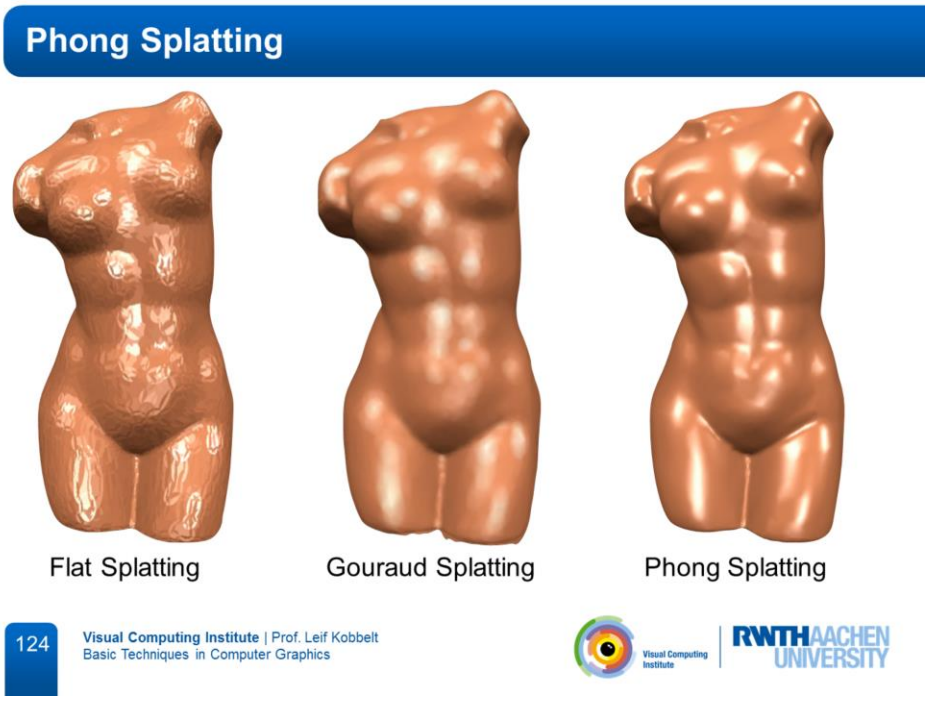
Splat Rendering

Flat Splatting Gouraud Splatting

Fading of the color across the splat will lead to gouraud splatting



Triangle-mesh rendering with different shading schemes.



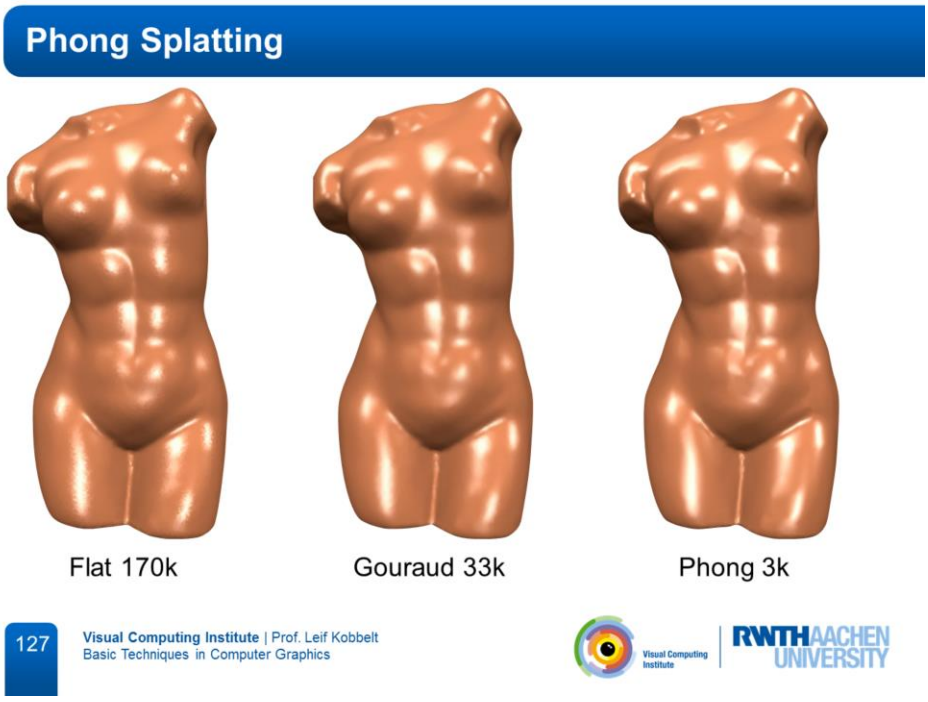
The equivalents for splat rendering
For phong splatting the normal gets rendered like the color in gouraud splatting. In a second rendering pass the per-pixel normal gets used to evaluate the phong model for each pixel.

Phong Splatting

- Flat
  - overdraw splats
- Gouraud
  - blend color values of overlapping splats (Gaussian kernels)
- Phong
  - linear normal fields
  - blend normal values of overlapping splats
  - deferred shading

Deferred Shading

- accumulate normal information (instead of color values)
- Gaussian blur
- evaluate shading per pixel (not per splat)



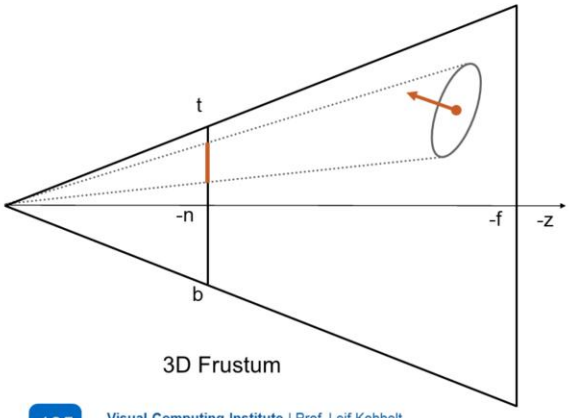
For similar quality, much more splats are needed in flat and even gouraud shading. The same was true for triangle meshes. The more complicated shading helps to reduce the mesh complexity and thus can to lead even higher speeds.

Rendering

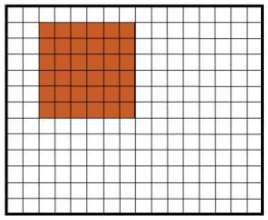
- Send *one* vertex per splat

Rendering

- Send *one* vertex per splat
- Determine projected size



3D Frustum



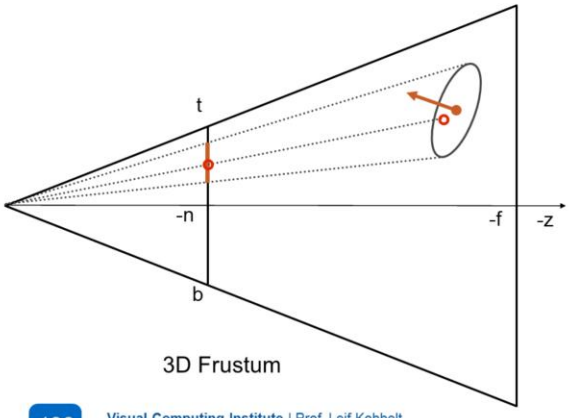
2D window

Visual Computing Institute | RWTH AACHEN UNIVERSITY

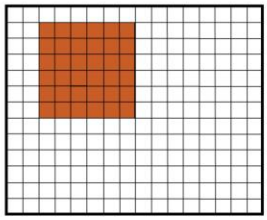
Better: just send one vertex that represents the center of the sphere, render it as a large point by OpenGL and discard fragments that don‘t belong to the splat in the fragment shader.

Rendering

- Send *one* vertex per splat
- Determine projected size
- Elliptical shape



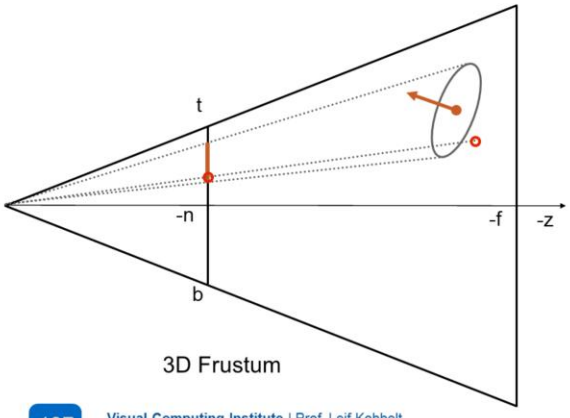
3D Frustum



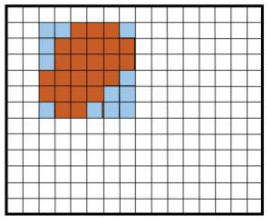
2D window

Rendering

- Send *one* vertex per splat
- Determine projected size
- Elliptical shape



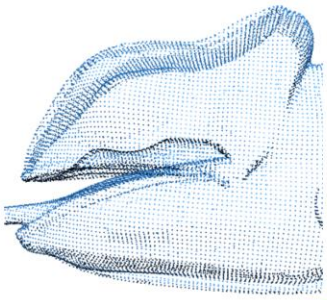
3D Frustum



2D window

GPU-based Splatting

1. Render each splat by one GL_POINT
   ⇒ Renders one vertex per splat

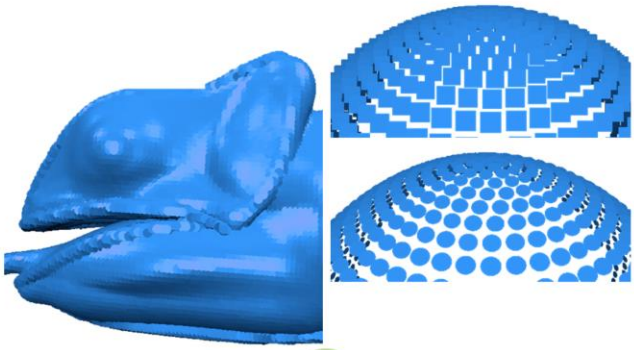


GPU-based Splatting

1. Render each splat by one GL_POINT
2. Adjust point size
   - perspective computation in vertex shader
   - image-space squares

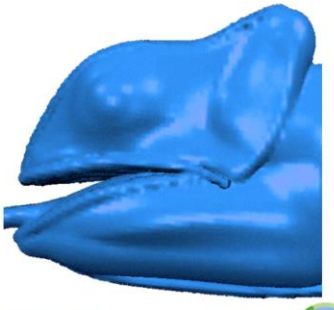
GPU-based Splatting

1. Render each splat by one GL_POINT
2. Adjust point size
3. Discard pixels outside splats
   - Compute/estimate in pixel shader
   - Image-space ellipses



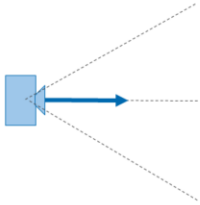
GPU-based Splatting

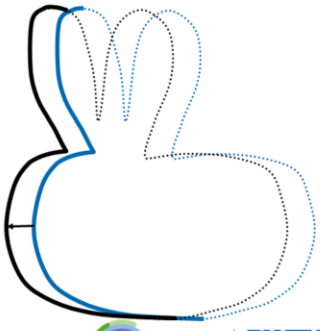
1. Render each splat by one GL_POINT
2. Adjust point size
3. Discard pixels outside splats
4. Blend overlapping splats



2-Pass Rendering

1. Draw geometry
   - fill z-buffer, don't update color buffer
2. Shift by ε in z-direction
3. Draw geometry
   - enable blending
   - don't update z-buffer





Depth Correction

- Blending depends on depth values
- Per-pixel depth correction necessary!