Object Space Volume Rendering

In object space rendering methods, the main loop is not over the pixels but over the objects in 3-space.

In the case of direct volume rendering, “objects” can mean:

- layers of voxels: image compositing methods
  - 2D texture based
  - 3D texture based
- voxels: splatting methods
- cells: cell projection methods

Texture-based volume rendering

Volume rendering by 2D texture mapping:
- use planes parallel to base plane (front face of volume which is “most orthogonal” to view ray)
- draw textured rectangles, using bilinear interpolation filter
- render back-to-front, using α-blending for the α-compositing

Volume rendering by 3D texture mapping (Cabral 1994):
- use the voxel data as the 3D texture
- render an arbitrary number of slices (e.g., 100 or 1000) parallel to image plane (3- to 6-sided polygons)
- back-to-front compositing as in 2D texture method

Limited by size of texture memory.

The shear-warp factorization

In general the image plane is not parallel to a volume face.

The shear-warp method by Lacroute allows to render an intermediate image in the base plane:
- transform to sheared object space by translating (and possibly scaling) the voxel layers
- render the intermediate image in the base plane
- warp the intermediate image
The view transformation ("modelview" in OpenGL) is an affine transformation, consisting of a rotation and a translation. Ignoring the translation, the 3x3 submatrix can be factorized:

$$ \mathbf{M}_{\text{view}} = \mathbf{W} \cdot \mathbf{S} \cdot \mathbf{P} $$

where:
- $\mathbf{P}$ is a permutation matrix mapping the base plane (front face of the volume most orthogonal to the center view ray) to the xy-plane
- $\mathbf{S}$ is the shear matrix
- $\mathbf{W}$ is the warp matrix

The shear is of the form

$$ \begin{bmatrix} 1 & 0 & s_y \\ 0 & 1 & s_x \\ 0 & 0 & 1 \end{bmatrix} $$

Hence, the shear matrix

$$ \begin{bmatrix} 1 & 0 & s_y \\ 0 & 1 & s_x \\ 0 & 0 & 1 \end{bmatrix} $$

where $s_x$ and $s_y$ have to be solved for from $\mathbf{M}_{\text{view}}$.

The warp is a 3x3 matrix, but effectively an affine transformation of the xy-plane. The third row of $\mathbf{W}$ is irrelevant while two zeros in the third column are required to make the warp independent of $z$:

$$ \begin{bmatrix} w_{10} & w_{11} & 0 \\ w_{20} & w_{21} & 0 \\ w_{30} & w_{31} & w_{32} \end{bmatrix} $$

Assuming for simplicity that $\mathbf{P}$ is the identity, we get:

$$ \begin{bmatrix} v_{10} & v_{11} & v_{12} \\ v_{20} & v_{21} & v_{22} \\ v_{30} & v_{31} & v_{32} \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & 0 \\ w_{20} & w_{21} & 0 \\ w_{30} & w_{31} & w_{32} \end{bmatrix} \cdot \begin{bmatrix} v_{10} & v_{11} & v_{12} \\ v_{20} & v_{21} & v_{22} \\ v_{30} & v_{31} & v_{32} \end{bmatrix} \cdot \begin{bmatrix} w_{10} & w_{11} & 0 \\ w_{20} & w_{21} & 0 \\ w_{30} & w_{31} & w_{32} \end{bmatrix} $$

It follows for the warp coefficients $w_j = v_j \quad (j \neq 2)$

for the shear coefficients

$$ \begin{bmatrix} s_x \\ s_y \end{bmatrix} = \begin{bmatrix} s_x \\ s_y \end{bmatrix} \cdot \begin{bmatrix} v_{10} & v_{11} & v_{12} \\ v_{20} & v_{21} & v_{22} \\ v_{30} & v_{31} & v_{32} \end{bmatrix}^{-1} \begin{bmatrix} v_{10} \\ v_{20} \\ v_{30} \end{bmatrix} $$

and for $w_{22}$ (not needed)

$$ w_{22} = -s_x v_{30} - s_y v_{31} + v_{32} $$

If $\mathbf{P}$ is not the identity, permuted versions of $\mathbf{S}$ and $\mathbf{W}$ can be used.

The same factorization can be used, but now in homogenous coordinates:

$$ \mathbf{M}_{\text{view}} = \mathbf{W} \cdot \mathbf{S} \cdot \mathbf{P} $$

The shear and scaling matrix $\mathbf{S}$ gets the form

$$ \begin{bmatrix} 1 & 0 & s_y & 0 \\ 0 & 1 & s_x & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

It does:
- a translation of $x$ by $s_x z$ and of $y$ by $s_y z$, followed by
- a scaling with $\frac{1}{1+s_z}$
The warp matrix $W$ is:

$$
W = \begin{bmatrix}
0 & w_{10} & 0 & w_{20} & 0 & w_{30} \\
0 & w_{11} & 0 & w_{21} & 0 & w_{31} \\
w_{00} & w_{12} & w_{02} & w_{22} & w_{03} & w_{13}
\end{bmatrix}
$$

The zero in the bottom row is needed to make the warp independent of $z$.

Assuming again that $P$ is the identity, we get:

$$
M_{\text{view}} = W \cdot S = \begin{bmatrix}
w_{00} & w_{01} & w_{03} & w_{02} \\
w_{10} & w_{11} & w_{13} & w_{12} \\
w_{20} & w_{21} & w_{23} & w_{22} \\
w_{30} & w_{31} & w_{33} & w_{32}
\end{bmatrix}
$$

It follows for the warp coefficients $w_j = v_j$ ($j \neq 2$) for the shear coefficients

$$
\begin{bmatrix}
s_x \\
s_y \\
s_z
\end{bmatrix} = \begin{bmatrix}
v_{00} & v_{01} & v_{03} & v_{02} \\
v_{10} & v_{11} & v_{13} & v_{12} \\
v_{20} & v_{21} & v_{23} & v_{22} \\
v_{30} & v_{31} & v_{33} & v_{32}
\end{bmatrix}^{-1} \begin{bmatrix}
v_{00} \\
v_{10} \\
v_{20} \\
v_{30}
\end{bmatrix}
$$

and for $w_{22}$ (not needed)

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

The shear-warp volume rendering algorithm is now as follows:

- For each voxel layer (parallel to base plane):
  - shear and scale the layer image by multiplying with $S$
  - apply transfer functions
- Generate intermediate image with $\alpha$-compositing
- Warp the image by multiplying with $W$

An advantage of this algorithm is that for scaling images a filter can be used to prevent undersampling (aliasing).

**Object space vs. image space**

Comparison of typical object space method (2D texture based) and image space method (raycasting).

Formally both are equivalent, only different nesting order of loops.

Practical differences:

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

**Object space vs. image space**

Post-classification can be done in graphics hardware:

Using (OpenGL) dependent texture (two texture mapping stages):

```
texture unit 0
(Interpolate scalar field)

texture unit 1
(apply transfer functions)

s

R,G,B,A
```

$x,y,z$
Preintegration is possible also in object space:
- Use slabs (space between two slices) instead of slices
- Dependent textures:
  - 1st stage: interpolate scalar field in front and back slice
  - 2nd stage: look up integrated transfer function

Object space vs. image space

Advantages of splatting:
- applicable to structured and unstructured grids
- other reconstruction filters than trilinear interpolation are possible, e.g. sinc filter
Original algorithm (Westover 1990):
- orthographic view, uniform grids → all footprints are translates of a template
Elliptical weighted average (EWA) splatting (Zwicker et al. 2001)
- ellipsoidal Gaussians as footprints
- perspective view, low-pass filter for antialiasing

Splatting
Raycasting: "What does each voxel contribute to a given pixel?"
Splatting: "What does a given voxel contribute to each pixel?"
Splatting as a brute-force method:
- pre-processing:
  - for each voxel \( x_i \) render (raycast) a field \( s(x_i) = \delta_i \)
  - store resulting footprint images
- main loop:
  - for each voxel \( x_i \) adjust footprint image to effective TF value
  - blend all footprint images of a voxel layer (*sheet buffer*)
  - do \( \alpha \)-compositing of layers

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

Computation of thick vertex:
- compute determinants \( d_i = \det(x_j, x_k, x_l) \) \( (i=0,1,2,3) \)
where \( x_j, x_k, x_l \) are the vertices of the \( i \)-th face, relative to camera position, ordered ccw on outside of face
- if number of positive determinants is
  - odd: class 1
  - even: class 2
- interpolation weights (for coordinates and data) of thick vertex
  - for class 1:
    - \( d_0 \) (example + + +)
    - \( d_1 \) (example + + +)
    - \( d_2 \) (example + + +)
  - for class 2:
    - \( d_0 \) (example + + +)
    - \( d_1 \) (example + + +)
    - \( d_2 \) (example + + +)

Cells are projected to triangle fans consisting of
- 1 thick vertex (projection of the common edge of the tetrahedra)
- 3 or 4 thin vertices (on the silhouette)

Original algorithm: triangle fan in the image plane
Improved algorithm: triangle fan in space:
- thin vertices keep original position
- thick vertex is set to midpoint of projected edge
Advantages:
- depth test can be used (allows volume rendering into a scene)
- viewing direction and field-of-view can be changed (for fixed camera position), keeping projection
Assigning opacities:
• 0 for thin vertices
• preintegrated TF for thick vertex

Assigning colors:
• look up color TF for thin and thick vertices

Visibility sorting:
• generate partial ordering of cells based on adjacent pairs
• break cycles (rare, small rendering error, alternative: split a cell)
• sort list of front cells by distance to centroid

Rendering of triangles with fragment program:
• interpolate \( s(x) \) for points on front and back triangle
• interpolate cell thickness
• lookup color and opacity in preintegrated TF

Back-to-front compositing
• cells must be depth-sorted
• possible without re-sorting: camera turn, zoom
• depth test (z-buffer) must be enabled
• additional (opaque) objects must be rendered before the volume

Example: Visualization of smoke propagation.
Simple smoke model (used in fire protection engineering):
• absorption proportional to \( s(x) \) (particle concentration)
• leading to simple preintegrated(!) opacity TF:

\[
\alpha = 1 - e^{-\frac{d}{\tau}}
\]

When compositing cells with low opacity, opacities are essentially added.
Adding many very small opacities (e.g. between 0/255 and 1/255) leads to quantization artifacts.
Options to reduce artifacts:
• compositing with 16 bits
• \( \alpha \)-dithering: instead of standard rounding
  \[
  x \rightarrow \left\lfloor x \right\rfloor + \left(x - \left\lfloor x \right\rfloor \right) \cdot \frac{1}{2}
  \]
  use randomized rounding
  \[
  x \rightarrow \left\lfloor x \right\rfloor + \left(x - \left\lfloor x \right\rfloor \right) \cdot \text{rand}
  \]
  (Predicates \( \geq \) understood as functions with values 0 and 1, “rand” being a random function with range \([0,1])

Example: Quantization artifacts without and with \( \alpha \)-dithering.

Hardware-assisted visibility sorting (HAVS, Silva et al. 2005) is a faster cell projection algorithm:
• requires 4 RGBA float buffers for storing per pixel 7 pairs of
  – scalar field value \( s \)
  – distance \( d \) to camera
• initial cell sorting done by CPU, based on centroids, results in \( k \)-nearly sorted sequence, with \( k \leq 7 \)
• main loop: draw all cell faces from back to front
• fragment shader
  – does exact sorting of buffered \((s, d)\) pairs
  – computes “thickness” of cell behind the pixel, \( s_d = d_1 - d_2 \)
  – does (preintegrated) TF lookup with \( s_1, s_2, \alpha \) and \( \alpha \)-compositing