Texture-Based Direct Volume Rendering

R. Daniel Bergeron
Department of Computer Science
University of New Hampshire
Durham, NH 03824

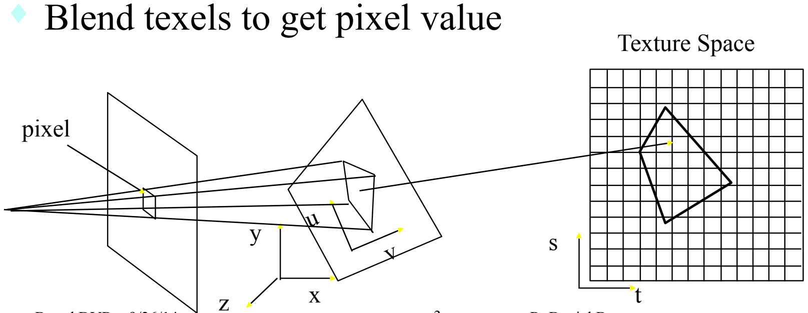
Based on:
Van Gelder and Kim, *Direct volume rendering with shading via 3D textures,* **Vis ’96**.
LaMar et al., *Multresolution techniques for interactive texture-based volume visualization,* **Vis ’99**.

Overview

- Hardware texture mapping is a great tool for dvr
- 2D texture mapping
  - create 2d slices through volume
  - map data in each slice to a texture with color and opacity
  - assign the texture to a polygon representing the slice
  - render the polygons (slices); let hardware composite textures
- 3D texture mapping
  - map volume data to a 3D texture with color and opacity
  - create rectangles parallel to screen, map these to positions in the 3D texture map and render with compositing

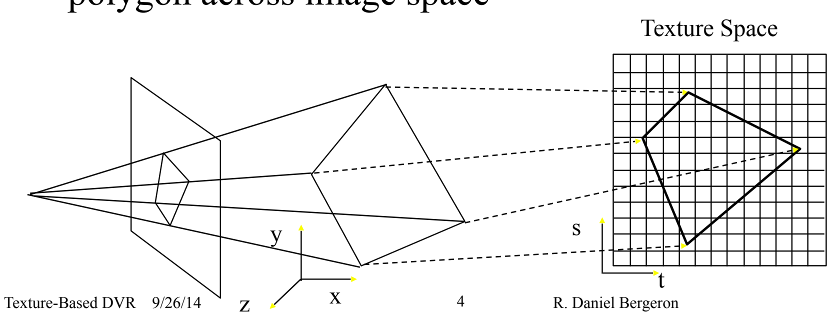
2D Texture Map Review

- Texture maps defined in *st* parameter space
- Associate a texture map with a polygon
  - define a mapping from 2D plane of polygon to texture space
- Map pixel to polygon plane and then to texture space
- Blend texels to get pixel value



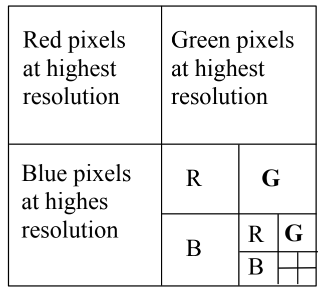
2D Texture Map Hardware

- Define polygon to texture space mapping by assigning texture coordinates to each vertex
- Map the polygon to the screen space
- Interpolate across texture space as scan convert polygon across image space



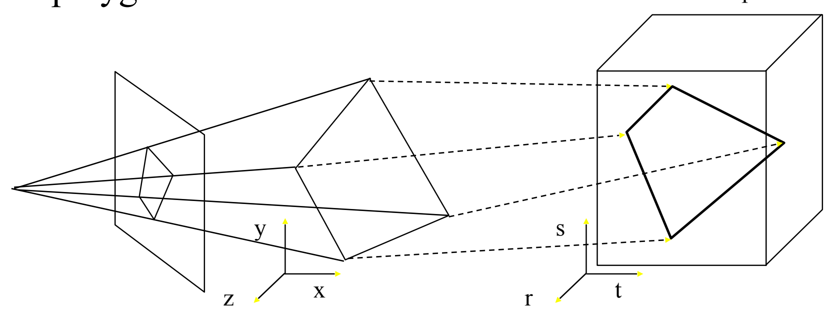
2D Texture Map Hardware (cont)

- Texture map support on graphics cards does most of the mapping and filtering
- Most boards today also implement hierarchical texture map called A-buffer
  - – Compress several resolution levels into a single texture map
  - – Hardware selects resolution level and does interpolation within and between resolutions
- Supports opacity and RGB



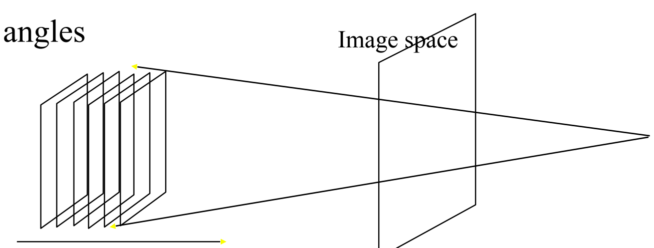
3D Texture Map Review

- Define an RGBA volume as a texture in *rst* space
- Map polygon vertices to the 3D texture space
- Interpolate through 3D texture space as scan convert polygons



Basic 2D Texture-Based DVR

- Original data slices are used to generate texture maps
  - – Use *xy* or *yz* or *xz* planes, depending on the view
    - » Generate 3 sets of texture maps as pre-processing step, or
    - » Dynamically regenerate texture maps as orientation passes 45° steps
  - – Each slice becomes a rectangle to render with its associated texture mapped to its surface
  - – Render from back to front
- Artifacts at large angles



Render from back to front

Voltx – Van Gelder & Kim

- Polygons always parallel to image plane
  - – fewer artifacts
  - – smoother transitions
- User-specified classification defines interior surfaces
- Add light source reflection from classified surfaces
  - – incorporate both reflection and ambient light into textures
  - – need to recompute textures if light source changes or if orientation of volume changes

Voltx: Creating Texture Maps

- Each *texel* in texture map corresponds to a voxel
  - it is a color and opacity derived from voxel data
  - combined ambient and reflective components
- Ambient component
  - "luminous gas" model, but only needs to integrate through a single "slab" of the volume (between two slices). Assume Δ for integration is constant (not really true for perspective)
- Reflective component
  - needs a classification algorithm to identify whether there is a reflection; if so, just add standard shading model, using surface normal (gradient)

Voltx: Classification

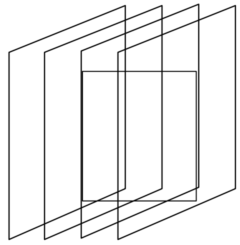
- User specifies a set of boundary values and a scale factor for each boundary that applies to a probability function in the region of the boundary.
- System uses the boundary value, the gradient at each point, and the scaled probability function to generate the weight for the reflective component.
- E.g., if bone is identified as 110 or higher and have a voxel of value 114 and gradient magnitude of 10, assume bone surface is .4 units from voxel in negative gradient direction. But, 130 with same gradient is not reflective.

Voltx: Computing Texel Values

- Texel values depend on light and orientation
- Need to change maps often, needs to be fast
- Build a lookup table for each material based on a quantized representation of the gradient.
  - Generate points on a sphere that are "evenly" distributed
    - triangular tesselation
    - icosahedron (12 vertices) and a dodecahedron (20 vertices) produce an initial set of 60 triangles that are recursively subdivided. They used 4 levels of recursion yielding over 7600 directions.
  - Quantize a gradient to one of these directions map it to an index into a table of colors for that material.

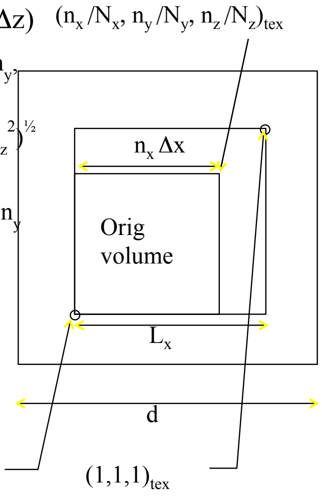
Voltx: Rendering Slices

- Create new volume centered at origin with sides = diagonal of original and slices parallel to image plane. Slices called a "proxy" geometry
- Create 3D texture map in new volume
- Force texture coordinates to range from 0-1 inside original volume
- Can define transformation that maps world volume coord (x,y,z) to texture coord (r,s,t)
- Rotations done by rotating texture map



Voltx: Defining the View

- Given volume is $n_x \times n_y \times n_z$ with spacing $(\Delta x, \Delta y, \Delta z)$
  - Let N_x, N_y, N_z be the least powers of 2 greater than n_x, n_y, n_z
  - Let $L_x = N_x \Delta x$, $L_y = N_y \Delta y$, $L_z = N_z \Delta z$, $d = (L_x^2 + L_y^2 + L_z^2)^{1/2}$
  - Map original llf corner $(-\frac{1}{2} n_x \Delta x, -\frac{1}{2} n_y \Delta y, -\frac{1}{2} n_z \Delta z)$ to texture (0,0,0) and original opposite corner $(\frac{1}{2} n_x \Delta x, \frac{1}{2} n_y \Delta y, \frac{1}{2} n_z \Delta z)$ to $(n_x/N_x, n_y/N_y, n_z/N_z)$



Voltx: Defining the View-2

- The mapping constraints are satisfied with
  $r(x) = (x + \frac{1}{2} n_x \Delta x) / L_x$
  $s(y) = (y + \frac{1}{2} n_y \Delta y) / L_y$
  $t(z) = (z + \frac{1}{2} n_z \Delta z) / L_z$
  - corners of bounding cube are at $(\pm\frac{1}{2} d, \pm\frac{1}{2} d, \pm\frac{1}{2} d)$ map these to texture coords using r,s,t above, will be *outside* the range (0,1)
- $(r,s,t) = (x,y,z) D^{-1} R^{-1} S T$
  where D^{-1} is uniform scale by 1/d, R is rotation of volume,
  S is scale by $(d/L_x, d/L_y, d/L_z)$, and
  T is a translation by $(\frac{1}{2} n_x \Delta x, \frac{1}{2} n_y \Delta y, \frac{1}{2} n_z \Delta z)$

Voltx: Rendering Planes

- Planes can have regions *outside* the volume
  - use OpenGL clipping planes so don't "render" empty voxels
  - user can redefine clipping planes to clip even more
- Number of planes
  - by default use $d/\Delta z$ planes; with default view each data point is sampled by one plane; 64^3 volume has 110 planes
  - more planes means more accurate images; they use 2-4 times default since extra planes aren't very costly

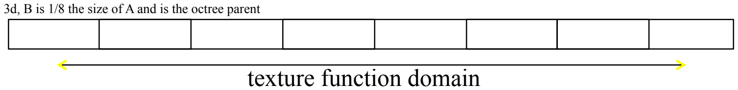
Multiresolution Volumes

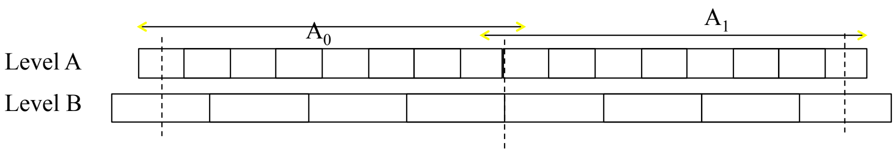
(from LaMar, Hamann, and Joy in **Visualization '99**)

- Given a *point of interest*, render close regions at high resolution, farther regions at coarser resolution
- Generate multiresolution texture hierarchy
- Generate octree representation of volume
- Given point of interest and viewing parameters, traverse the octree; at each node:
  - skip node entirely (subtree is outside viewing frustum)
  - render this node and skip all children
  - do not render this node, traverse its children

Generating Texture Hierarchies

- Textures are composed of *tiles*
  - linear interpolation used between tile centers, but texture function is undefined outside the centers of boundary pixels. 1-d example of 8-pixel tile, function defined over 7 pixels
- 2-level texture hierarchy – share boundary pixels
  - Image represented by A can be approximated by B with half the pixels and B can be the parent of A in a binary tree
  - In 3d, B is 1/8 the size of A and is the octree parent





Rendering: Selecting Tiles

- Traverse octree from root (coarsest level) down

  If tile of the current node is outside viewing frustum, return

  field-of-view criterion: select tile if it intersects the view frustum and tile's projected angle < field of view

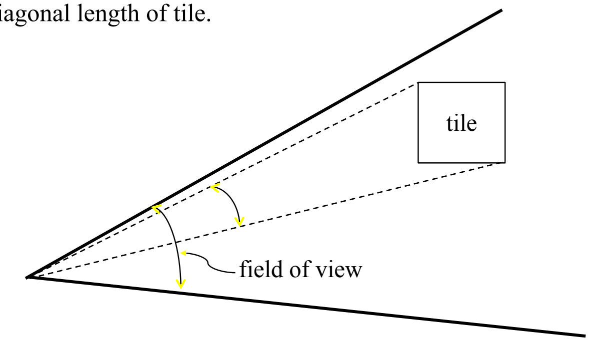
  distance criterion: select tile if distance from *point of interest* to center of tile > diagonal length of tile.

  If tile is selected

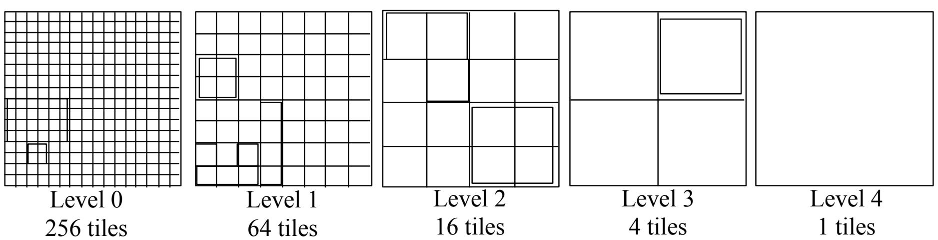
  render it

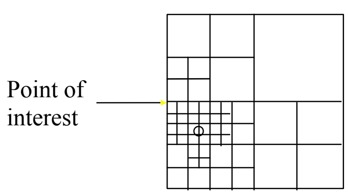
  else

  visit its children



Distance Criterion: 2D example





Proxy Geometries

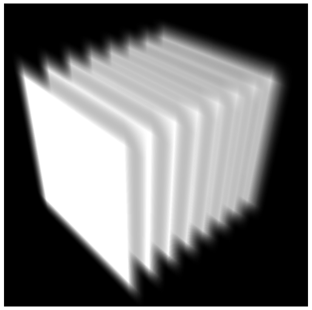
- Proxy geometry: the object to which textures are mapped
  - Object-aligned planes (OAP): original slices

    fastest, supported by lots of boards only 2D texturing needed, needs 3 sets of planes, light attenuation not correct if distance between planes not constant (worst at 45 degree angles)
  - Viewport-aligned planes (VAP): slices parallel to image plane

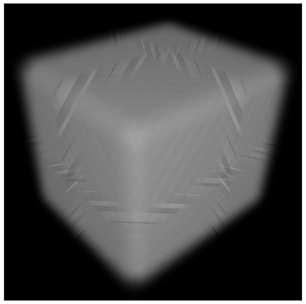
    only 1 3D texture, orthographic projections correct, not supported by all boards, more complex, still has some artifacts
  - Viewpoint-centered spherical shells (VCSS): concentric spherical shells centered at point of interest

    3D textures needed, no artifacts in perspective projection, slower

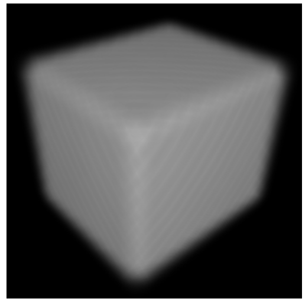
Proxy Visual Differences



Object Aligned Planes

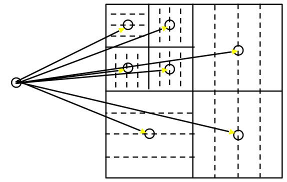


Viewport-Aligned Planes

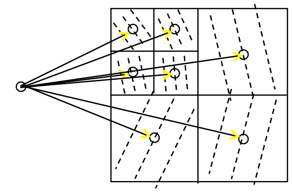


Viewpoint-centered
spherical shells

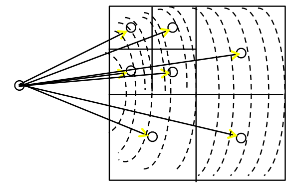
Multiresolution Proxy Geometries



Object-Aligned Planes



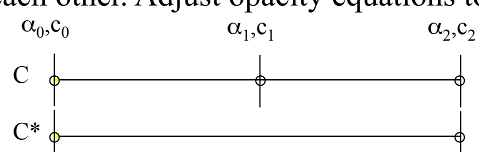
Viewport-Aligned Planes



VCSS

Preserving Visual Properties

- Varying resolutions introduce opacity properties
  - Traditional algorithms use equal step sizes along rays and integral approximation is based on step size
  - Samples along proxy geometries are at different distances from each other. Adjust opacity equations to compensate.



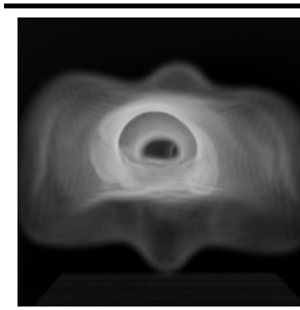
Can get approximately the same value for C and C* by letting

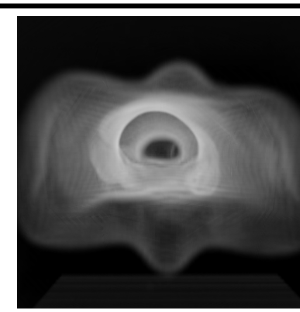
$$\alpha^* = 1 - (1 - \alpha)^2 = 2\alpha - \alpha^2$$

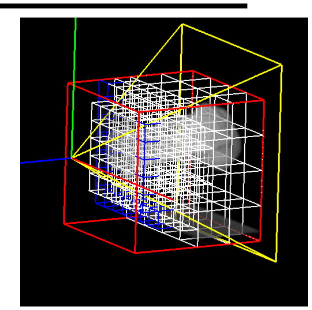
In general, if high resolution is k times resolution of low resolution:

$$\alpha^* = 1 - (1 - \alpha)^k$$

Sample Output







Horse metacarpus
fixed full resolution
VCSS
2.87 secs

Horse metacarpus
adaptive resolution
VCSS
1.53 secs