Interactive Volume Rendering Using 3D Texture-Mapping Hardware

Ray Casting Revisited

In the standard ray casting algorithm used for Volume Rendering, one or more rays are cast for every pixel of the image plane. All those rays are sampled at regular intervals, and the sampled color and opacity values are then combined using the standard compositing operator to approximate the volume rendering integral. Figure 1 shows several rays being cast through a two-dimensional cross section of a volumetric data set.

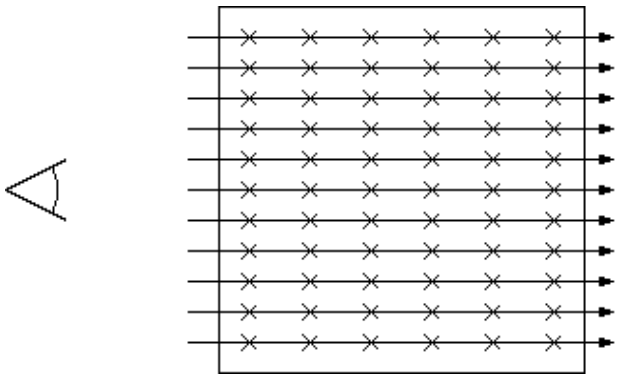

Figure 1: Rendering a volume by casting a set of rays through the volume and sampling each ray at regular intervals.

Though software implementations can use any appropriate filter to reconstruct the values at the sample positions, most implementations use tri-linear interpolation.

Looking at Figure 1 again reveals an alternate strategy to calculate the final image: Using a parallel projection, associated sample points of all rays are lying in planes slicing through the volumetric data set, see Figure 2. All these slices are parallel to each other and orthogonal to the viewing direction.

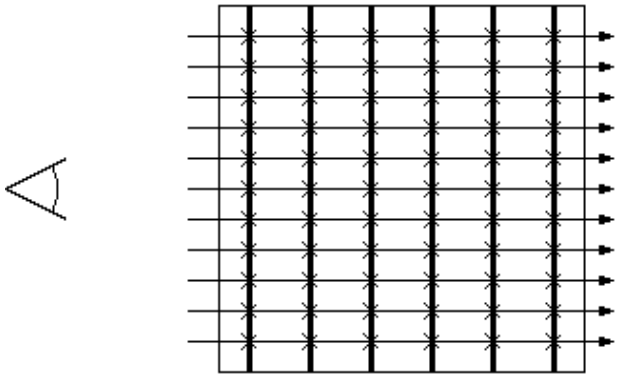

Figure 2: Associated sample points of all rays can be combined into planes slicing through the volumetric data set.

This means, that the standard ray casting algorithm can be replaced by an algorithm that proceeds in these steps:

1. For each slice, project the slice to the image plane and calculate the color and opacity values for each covered pixel appropriately.
2. Compose all the slices in either front-to-back or back-to-front order using the standard compositing operator.

The standard ray casting algorithm creates the final image by sampling all points contributing to a single pixel, and then combining those pixels into the final image. This method samples associated positions for all pixels in the image in parallel, and composes the generated images to create the final image. As it turns out, this approach is exactly equivalent to standard ray casting (unless precision is lost due to integer arithmetics in the compositing step).

2D Texture Mapping

Having rewritten the ray casting algorithm to be image-parallel, the question remains how to generate images of the independent slices, and how to calculate the correct color and opacity values for each sample position inside a slice.

To create images of the slices, one can use the standard texture mapping capabilities provided by graphics libraries like OpenGL. Figure 3 shows how pixel colors and opacities are calculated when rendering a texture-mapped polygon. In texture mapping, texture coordinates *(s, t)* are stored along with each vertex. These coordinates are interpolated across the polygon during scan conversion, and are used as coordinates for color look-up inside a texture image. Color and opacity values from the texture image are reconstructed using bi-linear interpolation.

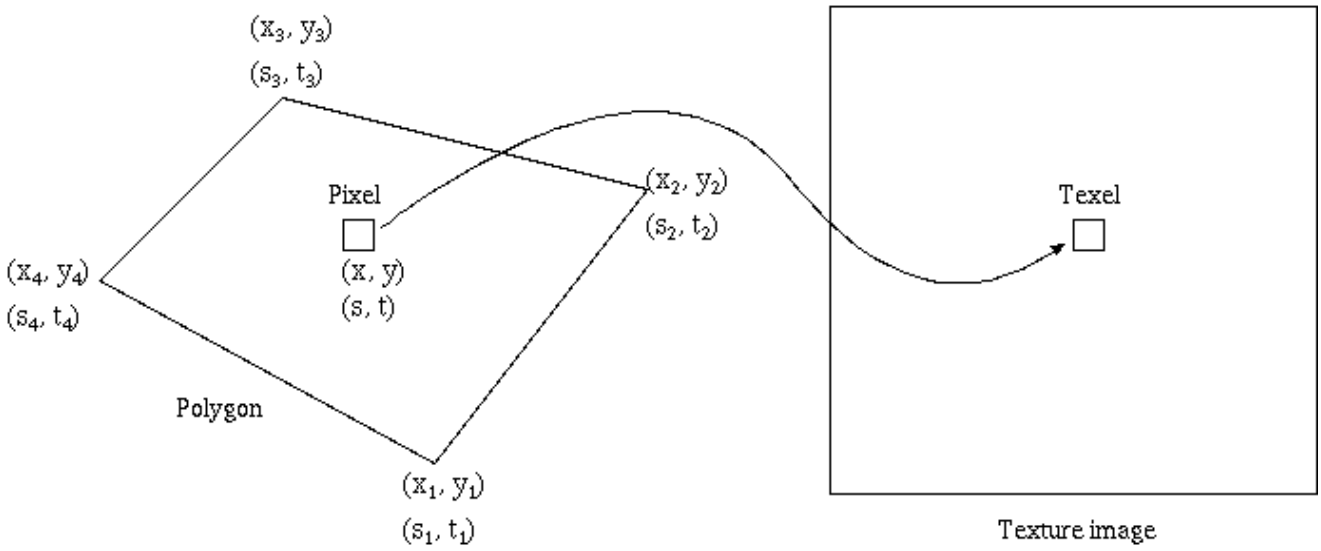

Figure 3: Calculating color and opacity of a pixel inside a texture-mapped polygon.

On modern graphics workstations, the complete process of interpolating texture coordinates and reconstructing texture values is incorporated into the scan-conversion process and done completely in hardware. Because texture mapping is such a common application, these routines are well-optimized and highly efficient.

To simulate ray casting using texture mapping, we have to slice the volumetric data set into parallel slices, and the use these slices as texture images to texture-map the projections of the sliced onto the image plane. To composite the slices, we can use the blending operations provided by the OpenGL graphics library. As it happens, one of the standard operators provided is exactly the compositing operator used in ray casting. Since compositing is also fully hardware-assisted, it is also very fast.

Treating Varying Viewpoints

So far, we have treated the special case of parallel projection and the view direction being parallel to one of the primary axes. How can we use texture mapping for arbitrary viewing directions?

There are two different methods to solve this problem.

1. One creates slice images for the given data set in a pre-processing step and uses different projection matrices to create arbitrary-viewpoint images of these slices. This is fast, since the (expensive) step of slice creation has to be done only once. On the other hand, image quality suffers because the views of all slices will be distorted due to the projection onto the image plane, see Figure 4. In the extreme case of the view direction being parallel to the slices, there will be no image at all.
2. One creates slices which are always orthogonal to the viewing direction, see Figure 5. This results in optimal image quality, but it means that the slice stack has to be re-generated for each new image. Furthermore, arbitrary ("oblique") slices through a cuboid are generally not square or rectangular, and can be anything from triangles to hexagons.

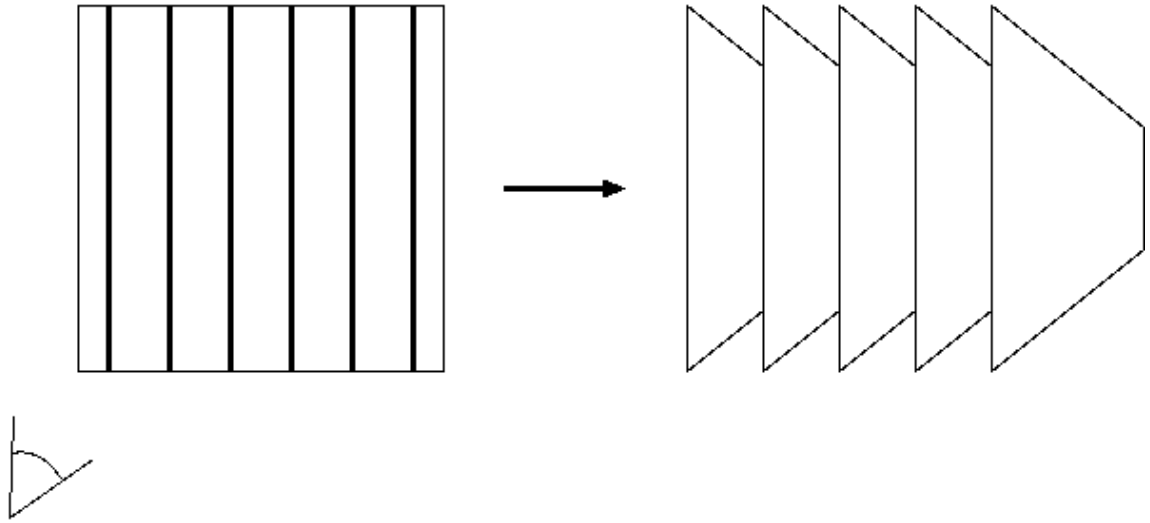

Figure 4: Slices parallel to a primary axis are distorted in arbitrary-viewpoint projections, degrading image quality. For illustration purposes, slices are drawn using a perspective projection to increase the effect.

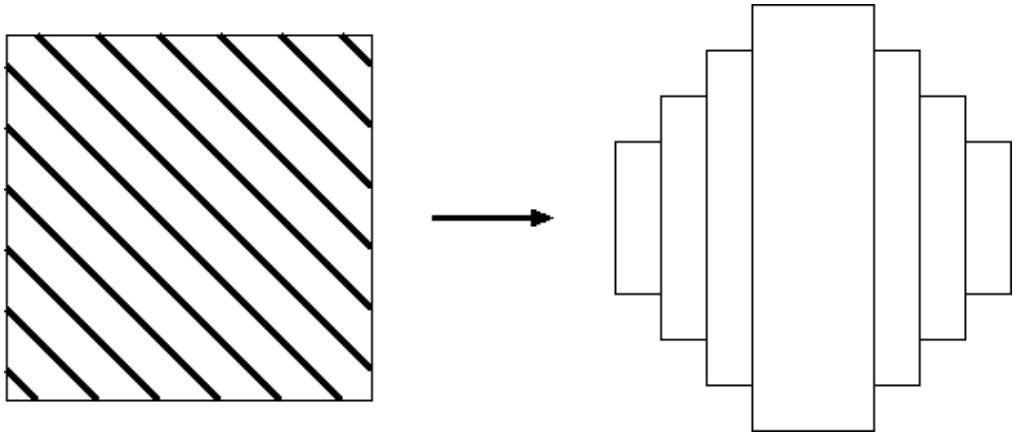

Figure 5: Slices orthogonal to the viewing direction are not distorted, but they have to be re-created every time the viewpoint changes, and they can have complex shapes.

3D Texture Mapping

To allow interactive generation of view-orthogonal slices, a special hardware technique has been developed. This is a generalization of texture-mapping to three-dimensional textures, appropriately called "3D texturing."

As seen in Figure 3, 2D texture-mapping interpolates two additional coordinates *(s, t)* across a polygon's interior. In 3D texture-mapping, *three* additional coordinates *(s, t, r)* are interpolated. To determine a pixel's color and opacity, these three coordinates are used as indices into a three-dimensional image, the 3D texture, see Figure 6. To reconstruct texture values, tri-linear interpolation is used.

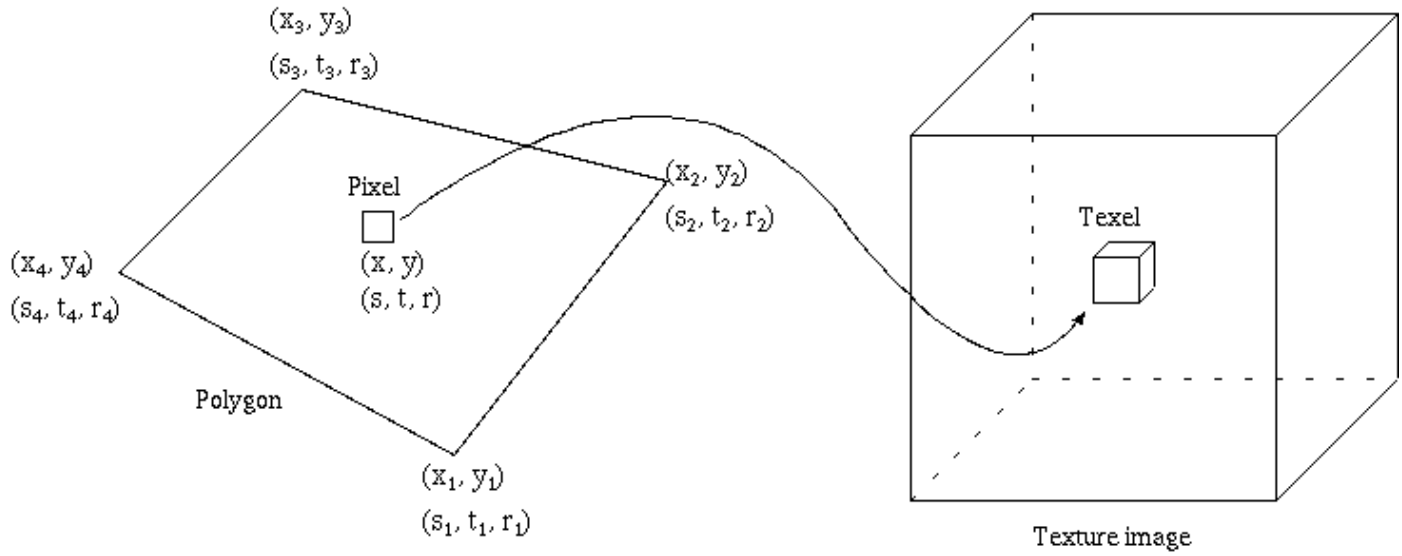

Figure 6: Calculating color and opacity of a pixel inside a texture-mapped polygon using a 3D texture.

3D textures allow direct treatment of volumetric data. Instead of generating a set of two-dimensional slices in a pre-processing step, the volumetric data is directly downloaded into the graphics hardware, and is directly used to calculate color and opacity values for each pixel covered by a rendered primitive.

Generating Oblique Slices

To generate oblique slices using 3D texturing, one only has to calculate the vertices of the slice, and then to generate the correct 3D texture coordinates for those vertices, see Figure 7. Then these coordinates are passed to the graphics library, and the graphics library will render the projection of the slice onto the image plane, using tri-linear interpolation to reconstruct each pixel's color and opacity values.

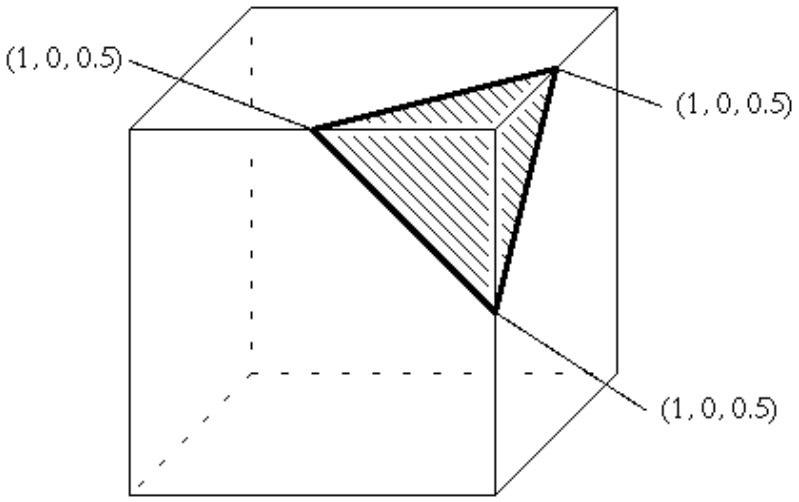

Figure 7: Calculating vertex coordinates and texture coordinates for an oblique slice. In this figure, the vertex and texture coordinates are identical.

How to Generate Stacks of Oblique Slices Efficiently

There is still a slight problem left in texture-based volume rendering: How to generate the stack of slices efficiently? A brute-force approach to slice generation would perform the following steps for each slice:

1. Calculate the plane equation for the plane containing the slice.
2. Generate a large-enough triangle that will contain the final slice.
3. Generate correct texture coordinates for the triangle's corners.
4. Clip the triangle to the volumetric data set using a standard polygon clipping algorithm.

There are several problems with this approach:

1. It is very inefficient. Each slice is treated independently, and clipping a triangle to a cube is a complex operation.
2. It is difficult to generate texture coordinates for the corners of the "large" triangle, such that the clipped texture coordinates will be exactly the needed ones. Numeric imprecision in the clipping process might introduce artifacts or degrade image quality.

To overcome these problems, I propose a better approach. This new approach is a generalization of the well-known scan-conversion algorithm for polygons, adapted to three dimensions. Each slice in the stack is treated as a 2D "scanplane," and the algorithm steps through the volumetric data set in negative view direction and creates a new scanplane from its direct neighbour exploiting scanplane coherence.

This "graph-based" approach generates slices directly with no need for additional clipping; it does not have precision problems, and it can be easily generalized to create stacks of slices through arbitrary convex polyhedra. With some more effort, it could also be used to slice arbitrary, non-convex polyhedra.

What About Perspective Viewing?

Until now we discussed texture-based volume rendering for the special case of parallel projections. Often it is desirable to generate perspective renderings of volumetric data, for example in Virtual Reality applications.

When ray casting a volumetric data set using a parallel projection, the sets of associated sample positions for each ray do no longer form view-orthogonal planes, but concentric spherical shells, see Figure 8. Though it is possible to create and draw these spherical shells using 3D texturing, it is expensive to do so. As it turns out, using view-orthogonal slices in the case of perspective projection is a good-enough approximation to the ray casting process. The only difference is, that the distances of sample positions along a ray are dependent on a ray's orientation.

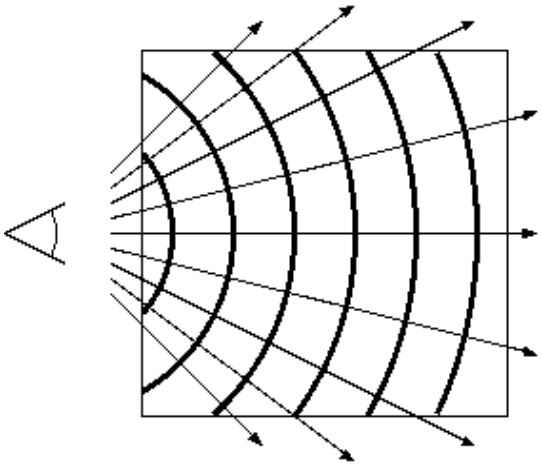

Figure 8: When using a perspective projection, associated sample positions do no longer form sets of view-orthogonal planes, but sets of concentric spheres.

Advantages and Drawbacks of Texture-Based Volume Rendering

Using texture-based volume rendering has the following advantages:

- **Speed**: Because graphics hardware is optimized for texture mapping, this technique allows for interactive frame rates even on "cheap" graphics boards found in today's game market.
- **Quality**: For parallel projections, the algorithm is an exact replacement for ray casting. If one is able to invest a bit more rendering time, to employ the floating-point precision of the accumulation buffer, image quality can be excellent.
- **Versatility**: Due to its high rendering speed, texture-based volume rendering can be used in a lot of interactive applications, like volume pre-viewing and Virtual Reality volume rendering ("$(VR)^2$" - sorry, couldn't resist to invent an acronym...).

Of course, texture-based volume rendering has some drawbacks as well:

- **Limited data size**: On today's graphics architectures, the size of 3D textures is limited, not only by the available texture memory, but also by implementation constraints. On an SGI InfiniteReality2 architecture,

the maximum data size available is 128^3 voxels. Also, on all current implementations the sizes of the data set are limited to powers of two in each direction. Though it is relatively easy to render larger data sets by subdividing them into "volume bricks," this approach is too slow to be used in interactive environments.

- **Limited reconstruction filters**: More severe, the only reconstruction filter offered by graphics hardware so far is tri-linear interpolation. Being a reconstruction filter of very poor quality, this limits the use of texture-based volume rendering in applications where high image quality is desired.
- **Regular grids only**: The rendering algorithm presented here does not work for non-regular grids.
- **No adaptive sampling**: Adaptive sampling can only be achieved by varying the distances between adjacent slices; inside a slice, however, the sampling density can not be changed.

Current Research: "Fake" Phong Illumination

In ray casting, the Phong illumination model is often used to provide the viewer with more visual clues to the structure of the data set rendered. The standard approach is to calculate the gradient of a scalar-valued volume data set at each voxel, and to evaluate Phong's lighting equation at each voxel as well.

In its standard form, the Phong equation depends on the following factors, see Figure 9:

1. The position *p* of the point to be lit,
2. the normal vector *normal* of the surface at *p*,
3. the direction *lightDir* from *p* to the light source, and
4. the direction *viewDir* from *p* to the viewpoint.

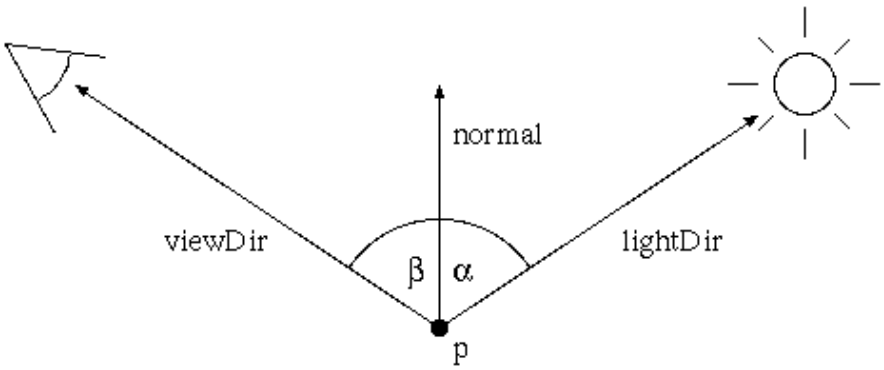

Figure 9: Parameters influencing the Phong lighting equation.

If one assumes that both the light source and the viewpoint are sufficiently far away from the points to be lit, the directions *lightDir* and *viewDir* will be (almost) constant for all points. In this "infinite viewer and light" approximation of Phong illumination (that is used as default by OpenGL), the Phong equation only depends on the normal vector of the surface at the point being lit.

Still, evaluating the Phong equation for each voxel of a volumetric data set is prohibitive for interactive environments. How can Phong lighting be further approximated, such that evaluating the equation at each voxel becomes unecessary?

The basic idea is to calculate the gradient vectors (which will be used as normal vectors) for each voxel in a pre-processing step, and then to quantize each gradient to a set of *N* pre-defined normal vector directions. Then it is sufficient to just store the index into the normal vector table with each vertex. Before rendering, the Phong equation is evaluated for each normal vector in the table, and the results are used to calculate a color map. The indices stored with the voxels are then used as indices into this color map during rendering, resulting in a decent approximation of real Phong illumination, see Figure 10.

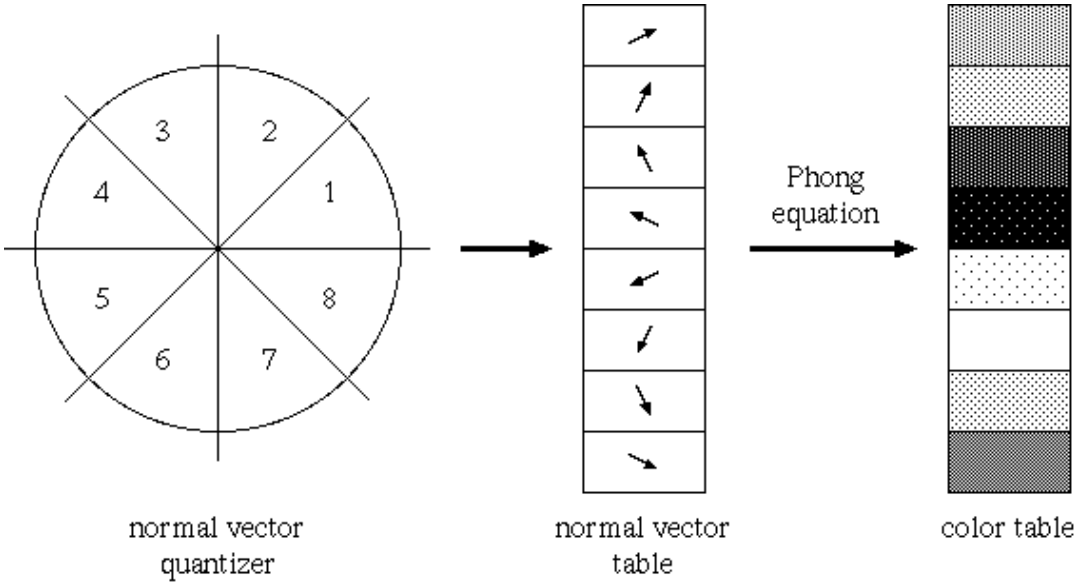

Figure 10: Quantizing normals and creating a color table for "fake" Phong illumination. This figure shows the two-dimensional case, where a normal vector quantizer divides the unit circle. In the three-dimensional case, a normal vector quantizer divides a unit sphere.

The number of normal vectors in the table is small (usually 240); evaluating the Phong equation for these 240 vectors can be done in real-time without any problems. Since current graphics hardware allows for color look-up during texture-mapping, using a color map to render the volumetric data is not an issue either.

The major problem of this approach is image quality. Because all gradients are quantized to one of (usually) 240 directions, the resulting image exhibits strong "directional aliasing". Creating good normal vector quantizers is essential to minimize the impacts of this principal problem on image quality.

The normal vector quantizer used in current implementations is static; it subdivides a regular dodecahedron into 60 triangles, and then each triangle into four, yielding a tesselation of the sphere consisting of 240 almost equal-sized triangles. Each normal vector falling into one of these triangles is quantized to the center of gravity of the respective triangle.

We are currently implementing a method to generate normal vector quantizers based on the volume to be rendered, by using an iterative optimization technique based on Simulated Annealing to improve an initial quantizer. This will - hopefully - not only generate a better and data-dependent distribution of normal vectors, but it also allows for having a different number of vectors. Hardware typically supports color maps having 256 entries; the standard quantizer only uses 240 of these.