Chapter 3

Visualization

Scientific (data) visualization is the process of creating a representation of an input data set that can be displayed on a visual output device like a CRT computer monitor, a printer or a multi-wall stereo projection system. Scientific visualization combines computer graphics, data processing, data analysis and computational geometry, numerical and discrete mathematics as well as a number of other disciplines. Even aspects from perceptional psychology and arts are important.

Visualization plays a key role in computer-based analysis of three dimensional data. Visualization techniques are also a basic prerequisite for any kind of interactive data treatment. Obviously interaction requires a way to perceive the subject of interaction, i.e. the data. At the same time a feedback on the effect of the interaction is needed. The predominant means of perceptual coupling between the computer and the user is the human visual system. In particular for image data, visualization is the most natural way of perception, although research is ongoing to use other output modalities like tactile devices or audio for data perception.

In this work we mainly have to deal with three different types of three dimensional data. Regular sampled scalar fields, triangulated surface models and graph structures, the latter potentially annotated with additional thickness information.

Before describing ways to visualize these types of data we briefly introduce some of the basic terms of computer graphics that will be needed in these sections. For a more comprehensive introduction we refer to a text book like [31].

3.1 Computer Graphics Basics

The most common type of computer graphics systems uses surface based descriptions of the three dimensional scenes that are to be rendered. For hardware accelerated rendering these surfaces typically have to be given as or converted to sets of polygons.

Surfaces can be characterized locally by a distinct outward normal vector $\boldsymbol{N}$. This normal vector plays an important role when describing the interaction of light with surface elements. In the following we will shortly review the popular reflection model of Phong. Let $\boldsymbol{L}$ denote the light direction, $\boldsymbol{V}$ the viewing direction and $\boldsymbol{R}$ the unit reflection vector (the vector in the $\boldsymbol{L}$-$\boldsymbol{N}$-plane

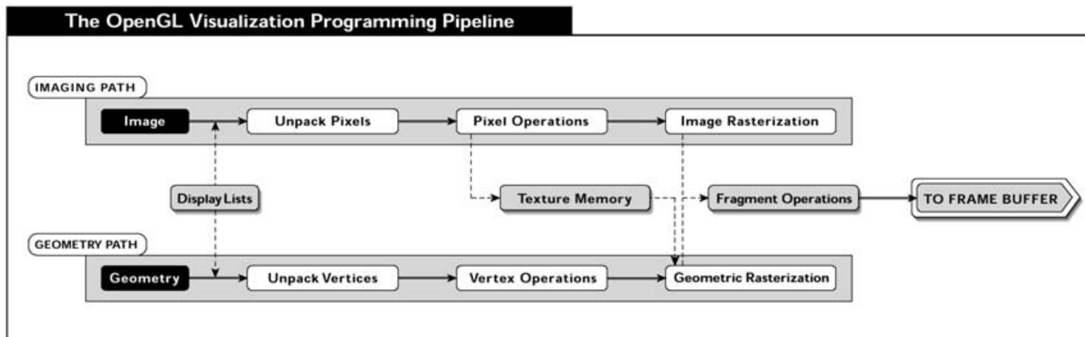

Figure 3.1: The basic components of the OpenGL pipeline. From http://www.sgi.com.

with the same angle to the surface normal as the incident light). Then light intensity at a particular surface point is given by

$$\begin{aligned} I &= I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}} \\ &= k_a + k_d \, \boldsymbol{L} \cdot \boldsymbol{N} + k_s \, (\boldsymbol{V} \cdot \boldsymbol{R})^n \end{aligned} \tag{3.1}$$

The first term, a global one, represents the ambient light intensity due to multiple reflections in the environment. The second term describes diffuse reflection due to Lambert's law. Diffuse light intensity does not depend on the viewing vector, i.e. diffuse reflecting objects look equally bright from all directions. The last term in Equation (3.1) describes specular reflections on a surface. Specular reflections or highlights are centered around the reflection vector $\boldsymbol{R}$. The width of the highlights is controlled by the exponent n, often called *shininess*.

Among others one can distinguish ray-tracing and z-buffer based rendering systems. In a ray tracing system light-rays are traversed backwards, i.e. from the observers eye through the image plane ("the monitor screen") until they hit an object. Then an illumination model like Equation (3.1) is evaluated. Possibly one (or more) rays are traced further to model reflection.

Most hardware accelerated graphics systems today are not ray-tracing but polygon/z-buffer based. They render the scene on a per-polygon basis. Occlusion is handled by a z-buffer storing the associated depth value for each pixel that has been rendered.

In typical polygon-based rendering systems Equation (3.1) is evaluated at each vertex and then interpolated within the polygon. The hardware does not require the normals at all vertices of a polygon to be the same. By using different normals at each vertex non-planar polygons can be visually approximated. The resulting color from the lighting calculation at the vertices is interpolated linearly in the triangle. This is often referred to as Gouraud Shading.

Instead of interpolating the result of Equation (3.1) within the polygon one could also interpolate the normal value and then evaluate Equation (3.1) per pixel. This yields much nicer shaped highlights even with coarse triangulations, but only the latest graphics hardware supports this up to now.

The most widely used graphics API for *professional* $3D$ graphics today is *OpenGL* [112]. As an API it specifies the data structures and procedure calls application programs must use to produce graphical output, as well as the semantics of the drawing process. The API design, however, is

very closely related to the design of the underlying hardware and for example the high end graphics systems from *Silicon Graphics* like the *InfiniteReality* [90] are dedicated OpenGL machines. The OpenGL pipeline is shown in Figure 3.1. Three dimensional geometry is specified by the vertex coordinates of the individual polygons and associated properties like color, surface normal, and texture coordinates. The vertices are projected onto the screen according to the current view point and view direction. Lighting calculations are performed at each vertex. Then each polygon is rasterized, i.e. it is computed which pixels are occupied by the projected polygon. Thereby the result of the lighting calculation and the texture coordinates are interpolated linearly. Finally, a texture lookup is performed and the pixel is written into the frame buffer along with its depth coordinate, in case that the depth value currently stored in the frame buffer does indicate that the new pixel is not occluded by an already drawn pixel. The two important performance bottle necks in this pipeline to be considered when designing an interactive visualization algorithm are the limited number of vertex transformations per second and the limited number of pixels that can be drawn per second. Depending on which of the limits is reached first, one refers to polygon-limited performance or fill-rate limited performance of a specific algorithm. An example for the first are complex iso-surface visualizations (Section 3.2.4), an example for the latter is texture based volume rendering (Section 3.2.5).

The pipeline drafted so far is only the most basic outline but it will be sufficient for the understanding of the techniques described in this work. Details especially on texture mapping and blending will be given when needed in the respective sections. For the variety of further extensions and details we refer e.g. to the documents provided at [20].

The development over the last ten years has shown that the rendering and shading model behind the initial OpenGL specification is not sufficient. More enhanced shading and rendering models as needed for the realistic modeling of car painture, for rendering shadows or the high-quality rendering of lines and transparent surfaces, which we describe in sections 3.3 and 3.4, are not directly supported. Using tricks like encoding the shading model into textures, using multi-pass rendering techniques, or with special OpenGL extensions, such enhancements can be implemented [44, 16, 9]. To increase the flexibility in the whole rendering pipeline, the specification for the newest versions 8 and 9 of Microsoft's $3D$ graphics API *DirectX* specifies a way for the programmer to define own lighting and texturing calculations, by providing little programs, called vertex shaders and pixel shaders. These programs are executed in the graphics processor itself. First hardware implementations exist on PC graphics boards [83]. Some of these capabilities are also available via extensions for OpenGL and they will become part of the OpenGL 1.4 and 2.0 specifications.

3.2 Image Data Visualization

The primary source data type for the algorithms discussed in this work are digital three dimensional image data sets, i.e. regular sampled scalar valued fields.

The scalar data values at each point can be mapped to gray-values or colors on the output device in a straight-forward way. The challenge is to reduce the dimensionality of the domain by converting the three-dimensional data set into a two-dimensional image suitable for display. Note that even if a three dimensional light field could be created with an appropriate output device like a holographic display, the problem would remain the same since the human retina can only record two

dimensional intensity distributions. Also with a stereoscopic display, which sends two different images to the two eyes, the problem remains for each of the two images.

The four most important techniques used for image data visualization are slicing, iso-surface rendering, maximum intensity projection, and direct volume rendering. In the following sections we will discuss these techniques and illustrate their use in the context of this work.

In the following we assume that a uniformly sampled image is given on a domain D as described in Section 2.1. With an appropriate interpolation scheme an image defines a scalar valued function:

$$f : D \longrightarrow \Re, \quad D \subset \Re^3 \tag{3.2}$$

3.2.1 Slicing

Slicing is the most commonly used technique to perform the dimension reduction mentioned above. Instead of visualizing the scalar field on the whole domain, only the data values on a subset are displayed.

The most commonly used type of subset is a planar cut, which is spanned by two perpendicular vectors $\vec{u}$ and $\vec{v}$ in the plane and one point in the plane $\vec{o}$. The two-dimensional image I representing the cut is given by

$$I(x, y) = f(x\vec{u} + y\vec{v} + \vec{o}) \tag{3.3}$$

with the range for x and y appropriately chosen. The function $I(x, y)$ can directly be displayed as a 2D image if an appropriate color mapping is applied (see next section).

This visualization technique is highly intuitive, especially if the scalar field represents light intensity as in the case of a confocal microscope. However, the reduction of information is tremendous. There are two common approaches to overcome this.

First, a time varying plane can be used to define a time varying image. For example if a plane perpendicular to the z axis is shifted along z in time, we would get

$$I(x, y, t) = f(\alpha_1 x, \alpha_2 y, \alpha_3 t), \tag{3.4}$$

where α_i are appropriate scaling constants. This way the third spatial coordinate that is missing on the display system is replaced by time.

Alternatively, the slices defined by multiple planes can be displayed simultaneously as shown in Figure 3.2. In order to visually encode the spatial relationship of the different slices the next step is to embed the images into a three dimensional scene and render this scene using a perspective projection as illustrated in the right part of Figure 3.2.

Slicing is not limited to planar subsets. Especially when dealing with multivariate functions like multi-channel confocal images one of the components can be used to define the geometry of the subset, while the other is mapped onto it. Figure 3.3 shows an example where the direction of a measured fluid-flow is used to compute a stream surface and the magnitude of the flow is extracted and displayed for each point of that surface.

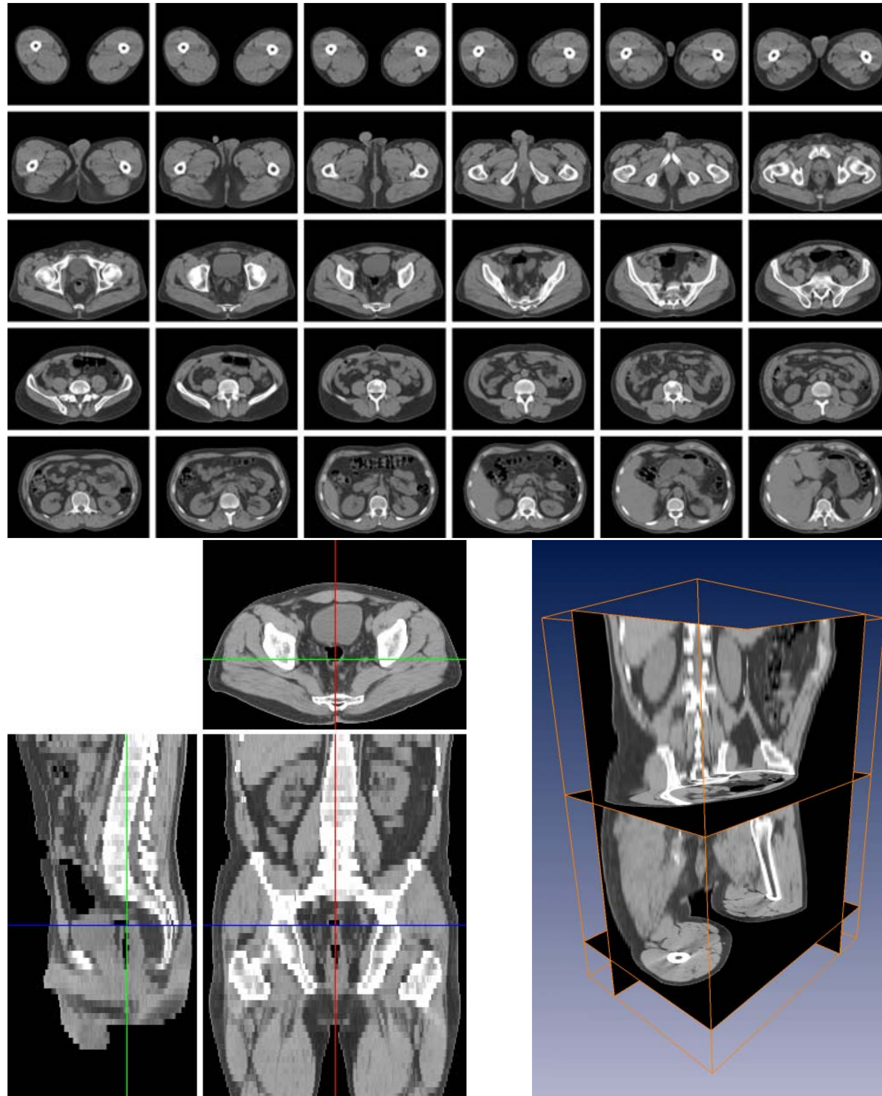

Figure 3.2: Multi slice visualizations. *Upper image:* Data set displayed in *virtual light box* style. The three dimensional relation has to be established mentally by the observer. *Lower left image:* Unlike traditional photos, digitally recorded image volumes allow orthogonal slices to be computed from the same data set. Note that each pair of images shares one coordinate axis. A crosshair allows the user to navigate. *Lower right image:* Multiple slices are visualized at their correct three dimensional position. Provided that the user can easily control the camera position, this is a very efficient and intuitive way for navigation and understanding of spatial relationships.

3.2.2 Color Mapping

Instead of mapping scalar values to shades of gray, they can also be mapped to colors using a so called transfer function. The range of applications for color mapping is broad. One simple reason to use color mapping, which should not be underestimated, is to make the images more appealing or more resembling to the images seen e.g. under a conventional microscope. Color can also

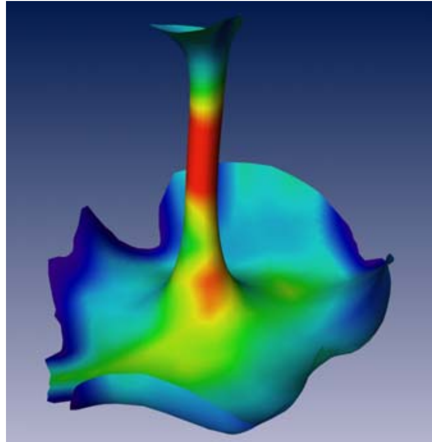

Figure 3.3: Non-planar slicing. A scalar valued field is evaluated on a non-planar geometry. Here a stream surface [121] is used as geometry to display a flow field's velocity magnitude.

help to allow an easier quantification and comparison of features that are not directly neighbored. Intended discontinuities in the color map can be used to indicate pixels with potential clipping, to find thresholds for segmentation, or to visually enhance iso-value lines or plateaus. Examples are shown in Figure 3.4. If multiple discontinuities are introduced as in the last image in 3.4, isolevels are visually enhanced. In the given example we only modified the luminance. This way visual degrees of freedom are saved and color hue or saturation could even be used to encode a further independent quantity. Note that is conceptually similar to the work described in [128], where C^1 discontinuities are used in elevation functions for height-field display.

3.2.3 Maximum Intensity Projection

While slicing is an appropriate first tool to visually understand the shape of structures with a significant volume, like the inner organs in the slices in Fig 3.2, it is much less well suited for thin structures, like the skin, or even line-like structures, like blood vessels or the dendrites of neuron cells.

The left part of Figure 3.5 shows a slice through a confocal recording of a single stained neuron. Most dendrites appear as small bright spots. Even with multiple slices or by varying the plane it is hard to understand the spatial relationship and structure of the dendritic tree. Only dendrites that lie almost in the plane can be well perceived.

If, as it is the case here, the structures of interest only fill a small fraction of the volume and if they appear significantly brighter or darker than the rest of the data set, then a maximum or minimum intensity projection can be a powerful tool. The basic idea is to project the data set onto a plane using an orthogonal or perspective projection and choosing along each projection ray the largest (or smallest) value of the input data set:

$$I(x, y) = max\{f(R(x, y, t))\} \qquad t \in [0, 1], \tag{3.5}$$

where $R(x, y, \cdot)$ is the projection ray through the image pixel (x, y).

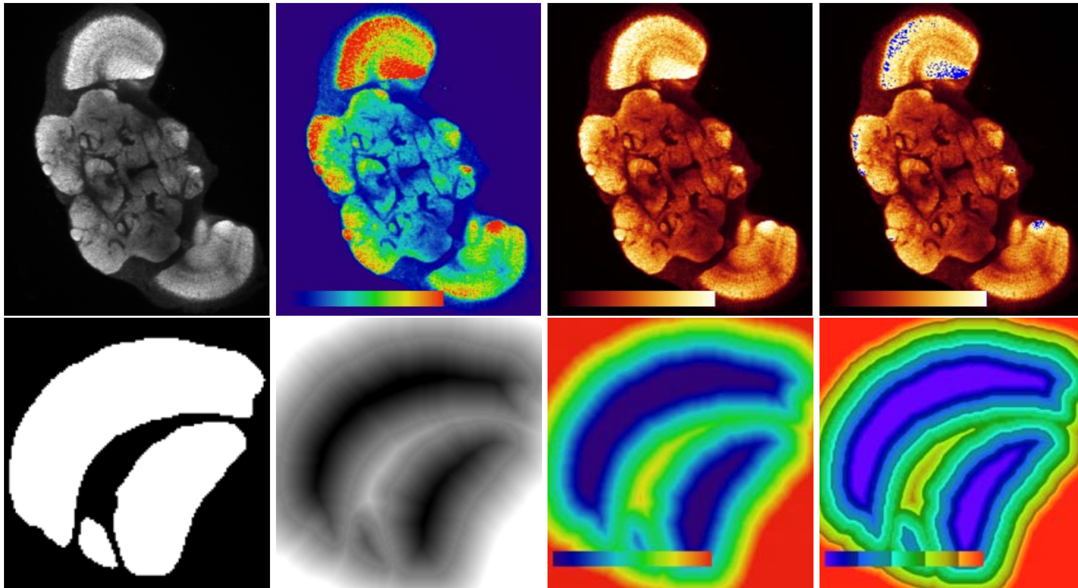

Figure 3.4: Visualization of scalar data on a slice using different mappings from data to color values. Top row: (i) Confocal scan of a Drosophila brain, linear mapping to gray values. (ii) and (iii) Different colormaps can be applied. The glow-style colors in (iii) resemble the colors seen in the microscope under certain conditions and are therefore considered pleasing. (iv) By setting all pixels equal or above a suitable threshold to some completely different color (here blue), thresholds can be found interactively or potential clipping can detected. Lower row: Visualized is a distance map computed from the labeling shown in (i). In the gray value mapping (ii) it is hard to quantitatively understand the data, which is easier using a colormap (iii). The quantitative expression of the image can be further enhanced by introducing discontinuities into the colormap (iv). Here we have modified the color luminance in a sawtooth fashion.

The right part of Figure 3.5 shows the maximum intensity projection of the same data set onto three perpendicular planes, giving a much better understanding of the topology and by means of the three orthogonal views still giving information about the three dimensional shape.

3.2.4 Iso-Surfaces

As can be seen in the images shown so far, the structures of interest are often of relative constant color, which is different from the surrounding background. We assume that an image is a continuous function. By extracting the level-set for an iso-value in between the two colors, one can therefore create a good representation of the objects. This can be understood as follows: The level-set for a given level (also called iso-value or threshold) of a scalar function in a three-dimensional domain is in general a two-dimensional manifold. If two neighboring samples of the digital images are on the opposite sides of the iso-value, the surface will in general pass in between them, at least if linear interpolation is used. Thus the level-set encloses the object.

One way to display an iso-surface is to compute a polygonal approximation of it. This can be done using the marching cubes algorithm proposed by Lorensen and Cline in [78]. The algorithm works on a per-cell basis. The cells are part of the grid, dual to the pixel grid: The cells are formed by eight neighboring image grid points. For each of the eight points it is determined whether it is

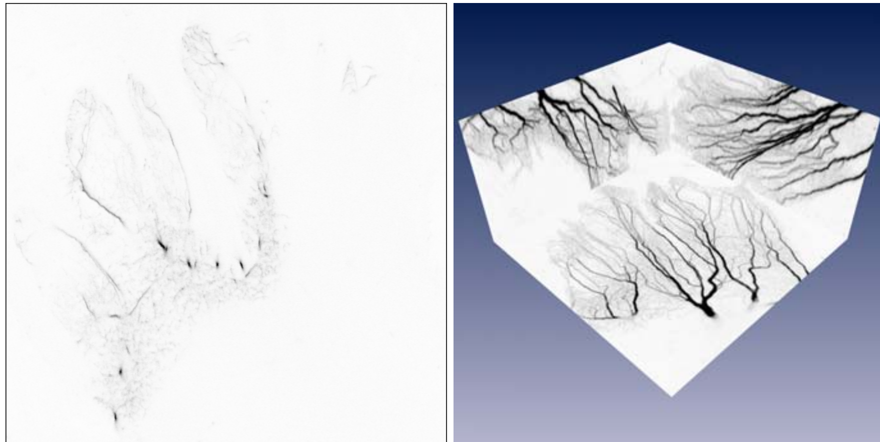

Figure 3.5: Slicing is not an appropriate tool for the visualization of line-like structures. While the overall shape is not perceivable in the left image, the three maximum intensity projections in the right image give a good impression. (Images inverted for better printing).

above or below the threshold. If both types occur for a cell, the isosurface must pass through this cell. The intersection points on the cell edges are determined using linear interpolation. Within the cell the surface is approximated with one or more triangles. Obviously only $2^8 = 256$ qualitatively different configurations can occur. Therefore, the triangulations can be stored efficiently in a look-up table. The triangulation for each configuration is generated manually, which is feasible since only 15 of the 256 cases are topologically different. Due to an inconsistent handling of some cases the triangulation originally proposed in [78] can lead to cracks in the surfaces. Once recognized this problem can be solved easily though [98].

Today iso-surface extraction is a standard technique for visualization of three-dimensional image data. Irrespective of its power to visualize certain aspects, as we will see throughout this work, a number of problems remain: Iso-surfacing requires that the structures to be visualized have an intensity range which does not overlap with the surrounding structures. The need to choose *one* threshold eliminates a huge amount of information contained in the data set. And especially for large or noisy data sets the number of triangles generated by the algorithm can be very large - too large to still be able to interactively rotate the objects. This problem can be addressed by adaptive simplification of the resulting surface in a post-processing step [111, 47, 57]. Another approach is to adaptively use a lower resolution of the input data in regions with less details already in the triangle generation step. For example an octree representation or other adaptively refined meshes can be used [114, 99, 136]. Such a re-organization of the domain can also serve to accelerate the extraction process itself [76]. Beside the design of a criterion that efficiently decides locally about the needed level of resolution, the challenge is to guarantee surface continuity in regions where cells from different resolution levels are neighbored.

The problem of iso-surface extraction is closely related to the general problem of geometry extraction from image data and image segmentations. Therefore, some more aspects and developments based on the marching cubes algorithm will be discussed in Section 6.1.

Isosurfaces can also be displayed without an explicit computation of a polygonal representation, as we will see in the next section [50, 135, 75].

3.2.5 Direct Volume Rendering

Another visualization technique for scalar valued data volumes is called *direct volume rendering*, or short *volume rendering*. Here the data volume is interpreted as a gaseous semi-transparent, shining cloud. Each point in the data volume is assigned a light emission and absorption property, depending on the data value at that point. Other effects like scattering and reflection can also be taken into account.

Linear Transport Theory

Using linear transport theory and neglecting wavelength dependencies one can derive a simple form of the equation of transfer [43]. The equation of transfer describes the change of intensity along a ray due to absorption, emission, and scattering:

$$\boldsymbol{n}\cdot\boldsymbol{\nabla} I(\boldsymbol{x},\boldsymbol{n}) = -\chi\, I(\boldsymbol{x},\boldsymbol{n}) + \eta, \tag{3.6}$$

where χ is the absorption coefficient and η is the light emission term. For volume rendering in scientific visualization applications these terms are determined by the data values by applying appropriate transfer functions. Typical transfer functions are transparent and less emissive for smaller data values (background voxels) and more opaque and more emissive for larger data values. By choosing different transfer functions, different aspects of the data set can be revealed. The design of a good transfer function can be a challenging task and in most cases manual interaction is involved. Work has been done to assist the user in this process as well as for fully automatic generation of transfer functions, describing opacity or opacity and light emission color, for example in [42, 56, 29]. If different data sets of the same type are visualized, e.g. confocal recordings of anti-body stained neuropil, experience shows, that sufficiently good results can be achieved by globally scaling and shifting a well chosen standard colormap.

Light absorption can be broken down into a "true absorption" term κ and an attenuation term σ due to scattering, $\eta = \kappa + \sigma$.

In an analogous way, the emission χ is split up into a light source term q and a term that describes the amount of light j that is scattered in the direction of the ray, $\chi = q + j$. The directional dependency of the scattering process can be described using a phase function $p(\boldsymbol{n},\boldsymbol{n'})$ that gives for a direction $\boldsymbol{n}$ of incidence the amount of light scattered in direction $\boldsymbol{n'}$. If the scattering particles are small compared to the wave length, the phase function resulting from Rayleigh scattering is

$$p = \frac{3}{4}(1 + \cos^2\theta). \tag{3.7}$$

In other cases often empirical models based on experimental results are used. See [43] for more details.

Using the phase function the light "emission" due to scattering in direction $\boldsymbol{n'}$ is

$$j(\boldsymbol{x},\boldsymbol{n}') = \frac{1}{4\pi}\int \sigma(\boldsymbol{x},\boldsymbol{n}')\, p(\boldsymbol{x},\boldsymbol{n},\boldsymbol{n}')\, I(\boldsymbol{x},\boldsymbol{n})\, d\Omega. \tag{3.8}$$

Inserting this into (3.6) we get

$$\boldsymbol{n}\cdot\boldsymbol{\nabla} I = -(\kappa+\sigma)\, I + q + \frac{1}{4\pi}\int \sigma(\boldsymbol{x},\boldsymbol{n}')\, p(\boldsymbol{x},\boldsymbol{n}',\boldsymbol{n})\, I(\boldsymbol{x},\boldsymbol{n}')\, d\Omega', \tag{3.9}$$

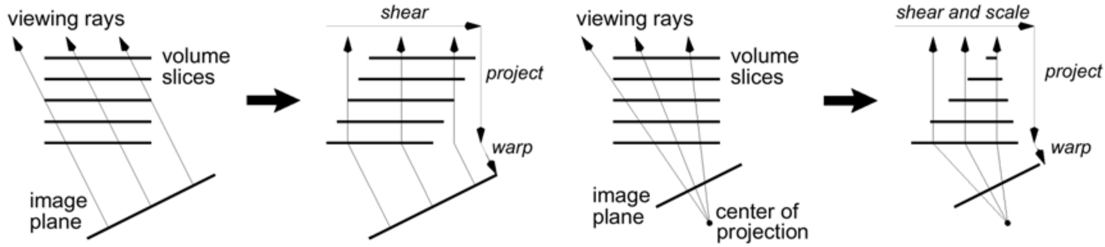

Figure 3.6: Shear-warp method for volume rendering. The left part shows the parallel-projection case, where the slices are first sheared, then composited and finally warped. In the perspective case, an additional per-slice scaling is added. (Figure adopted from [64].)

This is an integro-differential equation. It is often more convenient to rewrite the equation as a pure integral equation by integrating along a ray $\boldsymbol{x} = \boldsymbol{p} + s\boldsymbol{n}$. We introduce the optical depth between two points on the ray $\boldsymbol{x}_1 = \boldsymbol{p} + s_1\boldsymbol{n}$ and $\boldsymbol{x}_2 = \boldsymbol{p} + s_2\boldsymbol{n}$

$$\tau(\boldsymbol{x}_1, \boldsymbol{x}_2) = \int_{s_1}^{s_2} \chi(\boldsymbol{p} + s'\boldsymbol{n}, \boldsymbol{n})\, ds'. \tag{3.10}$$

and can easily derive ([43]):

$$I(\boldsymbol{x}, \boldsymbol{n}) = I(\boldsymbol{x}_0, \boldsymbol{n})\, e^{-\tau(\boldsymbol{x}_0, \boldsymbol{x})} + \int_{s_0}^{s} \eta(\boldsymbol{x}', \boldsymbol{n})\, e^{-\tau(\boldsymbol{x}', \boldsymbol{x})} ds'. \tag{3.11}$$

Note that the term $\eta(\boldsymbol{x}', \boldsymbol{n})$ still contains the integral over all possible directions of incidence. Therefore, it would be very complex to solve Equation (3.11) in full generality. Instead, in many cases the scattering terms are ignored or replaced by much simpler approximations, as we will describe below. For alternative derivations and a discussion of the various special cases of the volume rendering equation see [85].

Implementations

Even when neglecting the scattering effects, computing a full image according to Equation (3.11) is computationally expensive. For each pixel, the intensity for a ray that is cast parallel to the viewing direction and passes through that pixel has to be computed by discretizing the integral in Equation (3.11) appropriately. Assuming a typical window resolution of $512 \times 512 \approx 250,000$ pixels, it is not surprising that a straight forward implementation will not result in interactive frame rates. As soon as the viewing direction is not parallel to the major axis, following the ray through the data set and evaluating the input data using appropriate interpolation is expensive.

In 1994 Lacroute and Levoy have proposed a fast implementation based on a *Shear-Warp* algorithm [64]. This method first shears the data set, which is computationally inexpensive. In sheared object space the rays are parallel to one of the major axis directions. Computing the integral in Equation (3.11) is then just a matter summing up columns of pixels. In the final warping step an affine transformation is applied to the composited image, to produce the final image. The method is depicted in Figure 3.6. Using efficient data structures Lacroute and Levoy achieved rendering speeds in the order of one frame per second for a 256^3 data set at then current computers.

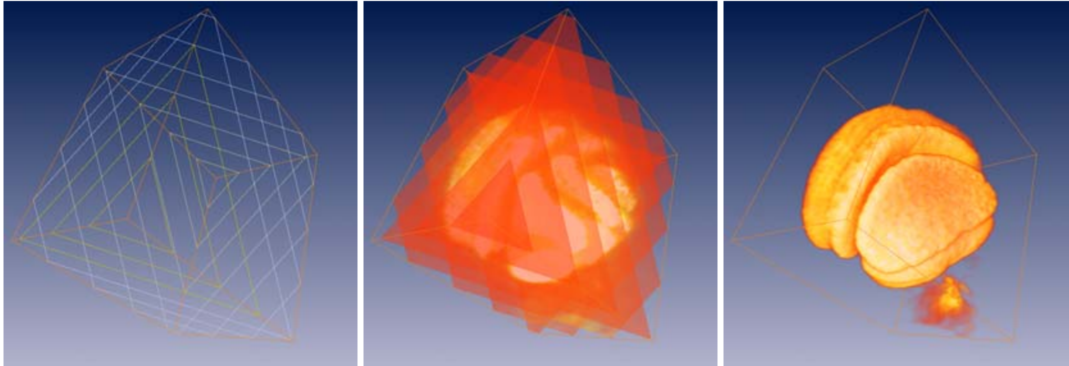

Figure 3.7: Texture based volume rendering. To render the image, slices are defined by intersecting equidistant planes with the data domain (left). The resulting polygons are rendered back-to-front with an RGBA (color+opacity) texture (middle). If a sufficient number of slices is drawn, a good approximation of the volume rendering integral can be achieved (right).

The first truly interactive volume rendering systems, which achieved several frames per second even for large data sets, however, used a different technique: They employed graphics hardware to compute the integral by repeated blending operations in the frame buffer. The data values were incorporated by using the three dimensional hardware accelerated texture mapping [2, 138, 23, 15].

The basic idea is to use the 3D data volume as a three dimensional texture map, which contains color and opacity values for each point in 3-space. Then a large number of polygons is drawn, which are perpendicular to the view direction. When applying the texture map, the texture hardware performs a tri-linear interpolation for each pixel after the polygons have been rasterized and uses the resulting color and opacity value. The method is illustrated in Figure 3.7. Today even many consumer graphics boards support three dimensional textures. If 3D texturing is not available, even 2D textures can be used. Then typically the polygons are not drawn exactly orthogonal to the view direction, but instead orthogonal to the major axis closest to the view direction. This way only three sets of polygons are needed for which two dimensional texture maps can be precomputed.

The above sketched algorithm is only the basic method for texture based volume rendering. Many improvements and extensions have been proposed over the last years. One important aspect for expressive images is reflection of light from an external light source on surfaces in the volume. This can be regarded as a special case of the scattering term in Equation (3.6). In [37] voxels are classified into reflecting or ambient, based on the magnitude of the data gradient. Large gradients occur at the boundary of materials, e.g. at the interface between bone and muscle tissue in the case of medical CT data sets. This way images very similar to polygon-rendered iso-surfaces can be computed, without the expensive generation and rendering of an explicit triangulation. In [37] the gradient direction is quantized and together with a material classification an index into a lookup table is computed, that allows fast lookup of the resulting color and opacity. Since the lighting calculation is view dependent, the lookup table has to be recomputed for each new view direction. The RGBA volume resulting from the lookup is fed into texture memory and rendered as described above. The drawback of this approach is that for each new view the full texture has to be recomputed and downloaded to texture memory, making the method slow. The authors report performance comparable to the shear-warp implementation in [64].

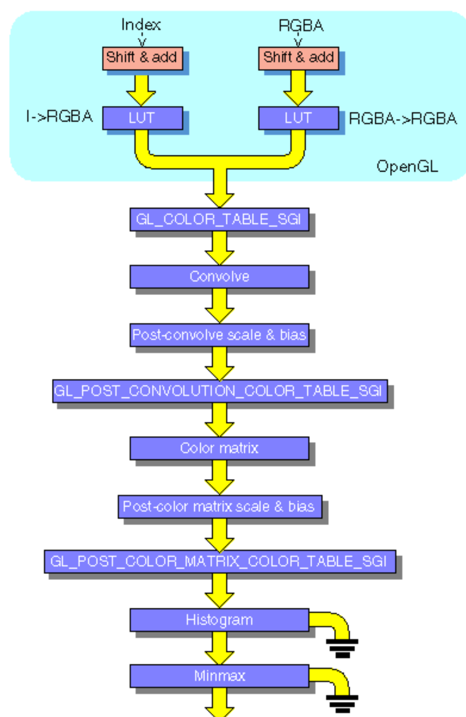

Figure 3.8: The imaging pipeline in the *Infinite Reality* graphics. The color matrix and the optional subsequent lookup to the POST_COLOR_MATRIX_COLOR_TABLE enable sophisticated operations like shading calculations in RGB space. This pipeline corresponds to the box labeled *Pixel Operations* in Figure 3.1. Figure adopted from [55].

With a graphics architecture like the SGI *Infinite Reality* [90] it is possible to take advantage of the imaging pipeline for the shading calculations in special cases. We briefly describe, how this can be done: The imaging pipeline consists of several stages that manipulate color values of pixels that are drawn into the frame buffer with `glDrawPixels`. The same manipulations can also be applied to a texture that is defined via `glTexImage3D`. Figure 3.8 shows the imaging pipe of the *Infinite Reality*. For shading based on the data gradient, the trick that can be used is this: Instead of real color (RGB) values, the three components of the gradient vector are written into the texture. These vectors are multiplied with the *Color Matrix*. If this matrix is initialized properly with the three coordinates of the light direction, this effectively computes the dot product between light and gradient. The subsequent lookup to the POST_COLOR_MATRIX_COLOR_TABLE allows to encode any shading model that depends solely on this dot product. This includes Phong shading in the special case of a head light, i.e. a light direction that is identical to the view direction. We will describe these techniques in more detail in Section 3.3.2. The drawback of this method is that the texture has to be sent down the graphics pipeline for each new view.

An efficient method to display iso-surfaces with texture based volume rendering, which avoids redefining the texture is described in [135]. Here the gradient is encoded in an RGB tuple. The alpha test is used to draw only the first visible pixel above a given threshold. The actual shading calculation is performed by applying the color matrix to each pixel's RGB value in a pixel copy operation. The color matrix was originally introduced to allow for color space conversions. Basi-

cally each row of the matrix is initialized with the three components of the light direction. Thus the product of the RGB vector with the matrix effectively computes the dot-product between light and gradient (which approximates the surface normal). This dot-product is proportional to the diffuse reflection according to Lambert's Law.

Westermann [135] also describes an alternative technique for shaded iso-surface rendering without explicit pre-computation of the gradient. Instead, the diffuse lighting term is computed by approximating a directional data derivative in direction of the light source with forward differences. These can be computed with frame-buffer arithmetic. Again, RGB tuples are interpreted as spatial vectors. In this case, however, color corresponds to the position of the surface points rather than to the gradient. The key tool for the actual shading calculation are pixel textures. They allow to use the RGB(A) color values of a pixel to be used as lookup indices into a three dimensional texture map. By rendering the image that contains the surface positions twice, the second time adding a small bias in light direction, the above mentioned directional derivative can be computed. The blend function has to be chosen so that the images are subtracted from each other.

Further extensions that are described in [135] are arbitrary clipping geometries implemented using the stencil test, rendering of data on spherical domains using pixel textures, and volume rendering of data on unstructured grids.

In summary, volume rendering using texture hardware has become a standard technique. It is fast, all newer graphics boards have texturing capabilities, many of them support three dimensional textures, and using the above described or similar methods, advanced effects can be achieved. Two problems which remain are the still limited amount of texture memory on many graphics boards and the lack of fast Phong-shaded multi-light source volume rendering. Both problems can be overcome using dedicated volume rendering hardware. MERL has developed a single-chip-based real-time volume rendering PCI board [100]. The authors reported a performance of 30 frames per second for a full Phong shaded volume rendering of 256^3 voxel data sets. A newer generation board, the *VolumePro 1000*, now distributed commercially by *TeraRecon Inc.*, supports rendering of data sets up to 2GB at interactive speed.

3.3 Visualizing Graphs and Lines

The next type of data to be visualized in the course of this work are lines or graphs. These are used to represent the dendritic tree of neurons or networks of vessels. Although rendering lines appears to be easy to achieve, the user gets confronted with serious problems when attempting it: On most common graphics workstations lines either have to be displayed using flat-shaded line segments, impairing the spatial impression of the image, or they have to be represented by polygonal tubes, strongly limiting the number of lines that can be displayed in a scene. In this section we describe an appropriate method for illumination of these lines and a way how to implement this model in a hardware accelerated way on standard graphics hardware. We have originally presented this method in the context of rendering field lines for vector field visualization [151].

It is a well-known fact that quality and realism of computer generated images depend to a high degree on the accurate modeling of light interacting with the objects in a scene. Shading effects are perhaps the most important cues for spatial perception. Consequently, much research has been performed to develop realistic illumination and reflection models in computer graphics. A widely

used compromise between computational complexity and resulting realism is Phong's reflection model described above (Equation (3.1)) which assumes point light sources and approximates the most important reflection terms by simple expressions [101]. Traditionally, this model is applied to surface elements. Today many graphics workstations offer hardware support for this kind of illumination.

The shading model can also be generalized to line primitives in $\mathbb{R}^3$. In the following we will make direct use of such a generalization. However, on current graphics workstations there is no direct hardware support for display of Phong-shaded line primitives. We achieve a fast and accurate illumination of line segments by exploiting texture mapping capabilities of current graphics hardware. Applying this new shading technique interactive frame rates can be achieved even for scenes with a large numbers of line segments. Taking light reflection on line primitives into account increases significantly spatial impression of the resulting images, and therefore is of particular significance for scientific visualization.

In scientific visualization the goal is not to render natural scenes in a photo-realistic way, but to generate images which provide maximal insight into numerical or experimental data. Nevertheless, shading effects are at least as important for the spatial interpretation of artificial images as in traditional computer graphics. Shading provides the observer with a minimum of realism in a world of cutting planes, isosurfaces, and symbols. Unfortunately there are a number of visualization techniques which are not based on surface primitives, and which therefore cannot make direct use of the hardware shading capabilities of current graphics workstations. As an example consider the various volume rendering techniques described above. While interactive frame rates can be achieved for simple emission-absorption models by exploiting graphics hardware, in general this is not yet possible if some sort of gradient dependent shading is included. Although rendering of line primitives is not as complex as volume rendering, the situation is similar. Traditionally, either flat shading has to be used or significant parts of the illumination calculation have to be computed without support by dedicated hardware.

After discussing illumination of line primitives in more detail, we show in Section 3.3.2 how it can be implemented using texture mapping techniques. In Section 3.3.3 we describe several visual extensions and enhancements, like use of color, transparency, and depth cueing.

3.3.1 Illumination of Lines in $\mathbb{R}^3$

Surfaces can be characterized locally by a distinct outward normal vector $\boldsymbol{N}$. This normal vector plays an important role when describing the interaction of light with surface elements. One popular reflection model is the Phong model given by Equation (3.1). The Phong equation applied to surfaces describes diffuse reflection by the dot product between surface normal and incident light direction. Specular reflection is modeled by the n-th power of the angle between reflection vector and view direction.

Let us now consider line primitives. In this case we can no longer define unique normal and reflection vectors. Instead, there are two-dimensional manifolds containing infinitely many possible normal and reflection vectors. Mathematically lines in $\mathbb{R}^3$ are said to have codimension 2. Fortunately common surface reflection models can be generalized to higher codimensions in a straightforward way. These generalizations have been discussed in detail by Banks [6]. For lines in $\mathbb{R}^3$ the results are quite obvious. From all possible normal vectors we simply have to select the

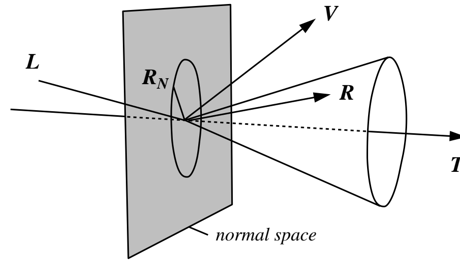

Figure 3.9: For line primitives there are infinitely many possible reflection vectors $\boldsymbol{R}$ lying on a cone around $\boldsymbol{T}$. For the actual lighting calculation we choose the one contained in the $\boldsymbol{L}$-$\boldsymbol{T}$-plane.

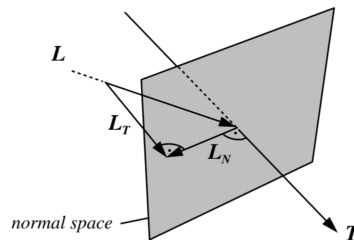

Figure 3.10: The light vector $\boldsymbol{L}$ can be decomposed into two orthogonal components $\boldsymbol{L}_T$ and $\boldsymbol{L}_N$ corresponding to the projection on the line's tangent and normal space, respectively.

one which is coplanar to the light vector $\boldsymbol{L}$ and the tangent vector $\boldsymbol{T}$. Taking this particular normal vector we compute the diffuse reflection term as for surfaces using Equation (3.1). Likewise, from all possible reflection vectors we choose the one coplanar to $\boldsymbol{L}$ and $\boldsymbol{T}$. Again, taking this particular reflection vector we use Equation (3.1) to compute the specular reflection term. The relevant vectors for line illumination are illustrated in Figure 3.9.

Instead of relying on a specially selected and explicitly calculated normal vector we would rather like to express diffuse light intensity for line segments solely in terms of $\boldsymbol{L}$ and $\boldsymbol{T}$. Therefore, we first project the light vector into the line's normal and tangent spaces, yielding an orthogonal decomposition $\boldsymbol{L} = \boldsymbol{L}_N + \boldsymbol{L}_T$. As illustrated in Figure 3.10, by applying Pythagoras' theorem we obtain

$$\boldsymbol{L}\cdot\boldsymbol{N} = |\boldsymbol{L}_N| = \sqrt{1 - |\boldsymbol{L}_T|^2} = \sqrt{1 - (\boldsymbol{L}\cdot\boldsymbol{T})^2}. \tag{3.12}$$

Using similar arguments we can express the inner product $\boldsymbol{V}\cdot\boldsymbol{R}$ responsible for specular reflection solely in terms of $\boldsymbol{L}$, $\boldsymbol{V}$, and $\boldsymbol{T}$, i.e. without referring to $\boldsymbol{N}$. First, observe that $\boldsymbol{R}_N = -\boldsymbol{L}_N$ and

$\boldsymbol{R}_T = \boldsymbol{L}_T$. We therefore have

$$\begin{aligned}
\boldsymbol{V}\cdot\boldsymbol{R} &= \boldsymbol{V}\cdot(\boldsymbol{L}_T - \boldsymbol{L}_N) \\
&= \boldsymbol{V}\cdot((\boldsymbol{L}\cdot\boldsymbol{T})\boldsymbol{T} - (\boldsymbol{L}\cdot\boldsymbol{N})\boldsymbol{N}) \\
&= (\boldsymbol{L}\cdot\boldsymbol{T})(\boldsymbol{V}\cdot\boldsymbol{T}) - (\boldsymbol{L}\cdot\boldsymbol{N})(\boldsymbol{V}\cdot\boldsymbol{N}) \\
&= (\boldsymbol{L}\cdot\boldsymbol{T})(\boldsymbol{V}\cdot\boldsymbol{T}) - \\
&\quad \sqrt{1-(\boldsymbol{L}\cdot\boldsymbol{T})^2}\sqrt{1-(\boldsymbol{V}\cdot\boldsymbol{T})^2}. \qquad (3.13)
\end{aligned}$$

Here we have replaced $\boldsymbol{L}\cdot\boldsymbol{T}$ by Equation (3.12). A similar expression has been used to rewrite $\boldsymbol{V}\cdot\boldsymbol{T}$.

3.3.2 Rendering Illuminated Lines

Despite the fact that the illumination equation looks the same for lines and surfaces, use of standard hardware shading techniques is impaired because for each new view or light direction a suitable normal vector has to be computed without utilizing graphics hardware. In the following we show how Eqs. (3.12) and (3.13) can be effectively evaluated using texture mapping capabilities of modern graphics hardware, thereby avoiding explicit normal vector computation. The technique allows us to achieve high frame rates even when large numbers of line segments have to be rendered.

Texture Mapping

We assume to have a graphics API available similar to OpenGL. In this graphics library at each vertex a homogeneous vector of texture coordinates can be specified. Usually the first components of this vector are taken as indices into a one-, two-, or three-dimensional texture map. A texture map may contain colors and/or transparencies which can be used to modify in various ways the original color of a fragment in the graphics pipeline. In addition it is possible to change texture coordinates using a 4×4 texture transformation matrix. This texture transformation is the key feature which makes it possible to employ texture mapping hardware for shading calculations.

Diffuse Reflection

Looking at Equation (3.12) we note that the diffuse light intensity of a line segment is a function of $\boldsymbol{L}\cdot\boldsymbol{T}$ only. Specifying a texture vector $\boldsymbol{t}_0$ equal to the line's tangent vector $\boldsymbol{T}$ at each vertex, this inner product can be computed in hardware using the following texture transformation matrix:

$$M = \frac{1}{2}\begin{pmatrix} L_1 & 0 & 0 & 0 \\ L_2 & 0 & 0 & 0 \\ L_3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

The first component of the transformed homogeneous texture vector $\boldsymbol{t} = \boldsymbol{t}_0 M$ then evaluates to

$$t_1 = \frac{1}{2}(\boldsymbol{L}\cdot\boldsymbol{T} + 1).$$

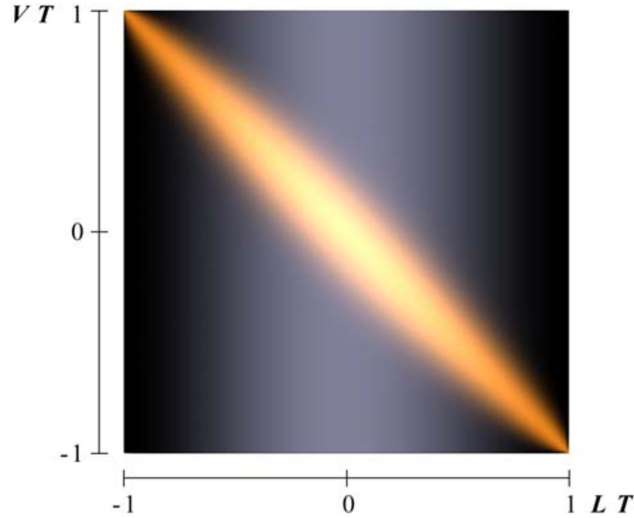

Figure 3.11: Two-dimensional texture map used to implement Phong's reflection model for line segments. Parameter values are $k_a = 0.1, k_d = 0.3, k_s = 0.6$, and $n = 40$.

Note, that t_1 always lies in the range $0 \ldots 1$. Therefore, this value can be used as an index into a one-dimensional texture map $P(t_1)$. The value of the texture map at location t_1 is chosen such that it resembles the diffuse light intensity corresponding to $\boldsymbol{L}\cdot\boldsymbol{T} = 2t_1 - 1$, namely

$$P(t_1) = I_{\text{diffuse}} = k_d \sqrt{1 - (2t_1 - 1)^2}. \tag{3.14}$$

Using a texture mode which takes the color of a line fragment to be equal to its texture color $P(t_1)$ we obtain an image which accurately shows line segments diffusely illuminated by a single point light source. If the light direction changes we simply have to update the texture transformation matrix. Vertices and texture coordinates of the line segments remain constant. This means that we can make use of OpenGL display lists to further increase rendering speed. Display lists allow one to specify multiple vertex and texture definitions using a single graphics library call.

Specular Reflection

The specular reflection term does not only depend on $\boldsymbol{L}\cdot\boldsymbol{T}$ but also on $\boldsymbol{V}\cdot\boldsymbol{T}$, as can be seen from Equation (3.13). To compute this additional inner product we initialize the second column of the texture transformation matrix with the current viewing direction:

$$M = \frac{1}{2} \begin{pmatrix} L_1 & V_1 & 0 & 0 \\ L_2 & V_2 & 0 & 0 \\ L_3 & V_3 & 0 & 0 \\ 1 & 1 & 0 & 2 \end{pmatrix}$$

While the first transformed texture component remains the same, for the second component we now get

$$t_2 = \frac{1}{2}(\boldsymbol{V}\cdot\boldsymbol{T} + 1).$$

In order to obtain the correct light intensity corresponding to $\boldsymbol{L}\cdot\boldsymbol{T} = 2t_1 - 1$ and $\boldsymbol{V}\cdot\boldsymbol{T} = 2t_2 - 1$ we use a two-dimensional texture map $P(t_1, t_2)$. Adding a constant ambient term k_a as well as the diffuse contribution from Equation (3.14) we can perform the whole shading calculation for a single light source in texture hardware. Figure 3.11 shows an example of a resulting two-dimensional texture map. One can clearly identify the highlight appearing at different angle positions on top of a diffuse background. If no highlight were present color would not depend on the viewing direction $\boldsymbol{V}$, as stated by Lambert's law.

It is worthwhile to note that there is an important special case, which allows one to use a one-dimensional texture even when specular reflection is present. This is the case of a headlight, i.e. a point light source located at the same position as the camera. In this case light vector and viewing vector are identical. Equation (3.13) simplifies to

$$\boldsymbol{V}\cdot\boldsymbol{R} = 2(\boldsymbol{L}\cdot\boldsymbol{T})^2 - 1.$$

Headlights are quite useful because they always guarantee an adequate illumination of the scene, irrespectively of the actual viewing direction. The user has not to bother with a tedious setup of light conditions. However, as will be shown later, also situations occur where other light positions are favorable.

Of course it is also possible to use the third column of the texture transformation matrix to compute an additional inner product. This would require the use of a three-dimensional texture map. Three different inner products would allow the illumination of lines by two point light sources located at arbitrary positions including specular reflection. Alternatively, one might discard specular reflection and instead introduce a third purely diffuse illuminating light source.

Excess Brightness

Banks [6] pointed out that there is a general problem when illuminating objects with codimension > 1. The overall intensity of an image increases and becomes more uniform, thus disturbing spatial perception. In case of lines in $\mathbb{R}^3$ this can be understood by the following consideration: We know that the normal vector is not a constant one, but is given by the projection of the light vector into the line's normal space. Choosing such a vector means minimizing the angle between light vector and normal. Therefore, in general the angle between these two vectors is smaller compared to the case of a fixed normal. This results in a more uniform brightness than we are used to perceive in real world. As suggested by Banks, we compensate the effect qualitatively by exponentiating the diffuse intensity term:

$$\hat{I}_{\text{diffuse}} = k_d\,(\boldsymbol{L}\cdot\boldsymbol{N})^p \tag{3.15}$$

In [6] a value of $p = 4.8$ was proposed. For the images in this paper we have used a value of $p = 2$, which produced nicer results.

3.3.3 Visual Enhancements

There are a number of ways to enhance and modify the rendering of field lines as discussed in Section 3.3.2. With color coding it is possible to depict an additional scalar quantity. Transparency can either be used to draw anti-aliased line primitives, to highlight particular regions in space.

Color

Color coding is a common method in visualization. Applying color to individual lines enables us to depict some additional scalar quantity. Ideally we would like to modify the curve's ambient and diffuse color components according to a given color lookup table. However, in our case color is directly taken from a texture map. Since we use the same texture map for all field lines it is not possible to set these components locally in a straight-forward way. Nevertheless, by using an alternative texture mapping mode it is possible to modulate, i.e. multiply, texture color with the object's base color. The latter can be defined for each vertex separately. This yields the desired effect with the restriction that also the specular highlight gets colored instead of remaining constant. In practice this has proven to be only a minor limitation.

Transparency

Transparency is a powerful concept which can be utilized in a number of ways. However, it requires geometric primitives to be rendered in a depth-sorted way. We will first discuss some applications of transparency, before we describe how to deal with the depth-sorting problem.

Anti-aliasing. Lines on a raster display may appear rather jagged, if a binary scan-conversion algorithm is used. These alias effects can be suppressed effectively by rendering pixels which are covered only partially by a line with an opacity proportional to the actual amount of overlap. This causes the final pixel color to be a mixture of the line's color and the color of the underlying object. Anti-aliasing of lines is directly supported in OpenGL. It improves image quality significantly. Jags tend to appear at different locations in successive frames with slightly different view directions. Since this is quite disturbing anti-aliasing is even more important for interactive applications and animations.

Highlighting. Transparency can be used to highlight important features of a scene. As with color in our application we can use an independent scalar field to define the transparency of a line at each vertex. Also drawing lines semi-transparently, allows us to depict a line thickness in the sub-voxel range.

Depth sorting. Drawing a transparent pixel of opacity α and color C causes the current color in the frame buffer to be updated according to

$$C_{\text{new}} = (1-\alpha)C_{\text{old}} + \alpha C. \tag{3.16}$$

In general if multiple transparent objects are present the final color depends on the ordering of the individual objects. Correct results are obtained using a back to front traversal. The situation is simplified if all objects are of equal color C. In this case all traversal orders yield the same result. This has been exploited by Max, Crawfis, and Grant [84], who applied flat shaded line bundles for vector field visualization. However, for illuminated lines color isn't constant. Therefore, individual lines have to be rendered in a depth-sorted way.

Figure 3.12: Shaded rendering of lines. The left two images compare flat shaded lines (i) and properly illuminated lines (ii). Especially when the line density is high, spatial structure is almost entirely perceived via the shading. Also for less dense images, like the dendritic tree of a neuron in the right two images, the shading (iv) improves perception, especially via the changing highlights when rotating the object (which is not visible in the static images depicted here of course).

In general it is impossible to achieve an exact depth ordering for extended curves in 3D, because mutual coverings may occur. Therefore, we split each line into many small line segments, which are sorted and rendered individually. To avoid resorting line segments each time the view direction changes, we use the following simplified algorithm: Three lists of pointers to field line segments are created. The lists are sorted in order of increasing x-, y-, and z-coordinates, respectively. During rendering the list that most closely resembles the viewing direction is traversed, either from back to front or from front to back. Although this method is not exact, it produces excellent results which can not be distinguished visually from the exact images. Experiments have shown, that typically only about 1% of all pixels receive slightly incorrect color values.

3.3.4 Results

We have presented a method for the illumination of lines and shown how it can be implemented efficiently in graphics hardware. With this method it is possible to render lines with proper illumination without performance degrade. Line illumination significantly improves expressiveness of the resulting images in many cases. Examples are shown in Figure 3.12. The shading also improves image quality when line and surface models are combined, as in Figure 5.7.

3.4 Enhanced Transparency

The fundamental problem when visualizing a volumetric data set with surfaces that have been extracted from the data set is occlusion. An outer surface like the skin in a medical data set will inhibit the perception of all inner organs. A way to tackle this problem is the use of transparency.

Although there are many interesting applications of transparency in computer graphics and visualization, the results of using transparency on standard polygon-based rendering systems often are quite disappointing. Especially when the transparency is large (opacity $alpha$ in the range of 0.1 to 0.3), the object's shape is hard or nearly impossible to perceive. The impression of such images is often that of a thin slab, instead of a three dimensional object, as we will see in the Figure 3.14.

The key observation is that transparent objects are mainly perceived by the attenuation of objects (or background pattern), which are placed behind them. Although there have been some early works [Gb78], in which the amount of transparency has been modulated in a view dependent way, standard polygon based rendering environments assume the transparency to be a constant material property. Then of course the attenuation only depends on the number of triangles that lie between the observer and a particular background point (for closed convex objects this number is always two). Therefore, the attenuation only displays the object's contour, but not its full three dimensional shape.

The shape of opaque objects is perceived through the spatial variation of its color by lighting models. When transparency is assigned to the model, this color typically is multiplied by a factor α, called opacity, before a pixel is combined with the background. For larger transparency values α becomes smaller and at the same time the absolute intensity variations become smaller. The observation is, that for typical α-values like $\alpha = 0.1...0.3$ the intensity variations do not suffice to provide a good three dimensional shape perception.

In the following we will show how the problem can be solved, by introducing a view dependent transparency term. This improves both: shape perception as well as quality and realism of the resulting images. The proposed model is easy to implement and it will be shown that all additional calculations can be implemented in a fully hardware accelerated way on standard OpenGL graphics hardware.

Section 3.4.1 explains why such an approach is quite natural and derives some mathematical expressions. Section 3.4.2 describes the implementation on OpenGL-like APIs, and discusses performance issues. It will be shown how texture hardware can be exploited to completely avoid performance drawbacks.

3.4.1 The Physical Model

Light traveling through a non-opaque medium is influenced in several ways. The most important effects are emission, absorption, and scattering. In the following we will focus on emission and absorption.

Absorption

The amount of light absorbed at a particular point will in general be a function of the total light intensity $I(x)$ and the material properties at that point. This can be expressed by

$$\frac{dI}{dx} = f(x, I(x)) \tag{3.17}$$

Often it is justified to assume f to be linear in I. Then equation (3.17) becomes

$$\frac{dI}{dx} = -\kappa(x)I(x), \tag{3.18}$$

where $\kappa(x)$ is the so called absorption coefficient. For many cases $\kappa(x)$ is constant within one material.

In this case equation (3.18) can be solved to be:

$$I(x) = I(0)e^{-\kappa x} \tag{3.19}$$

In other words: The light ray passing through a layer of thickness d is attenuated by a factor

$$\theta = e^{-\kappa d}. \tag{3.20}$$

θ is called the transparency of this layer. Note that d is the materials' extend in the light ray's direction. If the view direction is not perpendicular to the surface, then d is to be chosen as

$$d(\phi) = \frac{d_0}{\cos \phi},$$

where ϕ is the angle between the surface normal and the view direction, and d_0 is the layers original thickness. Introducing the surface normal $\boldsymbol{N}$ and the view direction $\boldsymbol{V}$ into Equation (3.20), we obtain

$$\theta(\boldsymbol{V} \cdot \boldsymbol{N}) = e^{-\kappa \frac{d_0}{|\boldsymbol{V} \cdot \boldsymbol{N}|}}.$$

or with $\theta_0 \equiv e^{-\kappa d_0}$:

$$\theta(\boldsymbol{V} \cdot \boldsymbol{N}) = \theta_0^{\frac{1}{|\boldsymbol{V} \cdot \boldsymbol{N}|}}. \tag{3.21}$$

Emission

In addition to the attenuation of light by absorption, the material can add light to the beam. If the light source density at a particular point emitting in direction of the observer is given by $q(s)$, then using the above results, we find that the intensity of emitted light, leaving the material after thickness d is

$$I_e(d) = \int_0^d q(x')e^{-\kappa(d-x')}dx'$$

The factor $e^{-\kappa(x-x')}$ takes into account that the light emitted at a particular point is partly absorbed again on its following way. For a layer of thickness d with constant light emission density viewed under a certain angle, using the notation and results from above, we get:

$$\begin{aligned} I_e(d) &= \frac{q}{\kappa}\left(1 - e^{-\kappa d}\right) \\ I_e(d) &= \frac{q}{\kappa}(1 - \theta) \end{aligned} \tag{3.22}$$

Reflection

In addition to the effects described above, transparent surfaces can reflect light that originates from a light source and directly strikes the surface. In a lighting model like *Phong's* model diffuse and specular reflection are distinguished. The diffuse reflection term describes the light reflected back after multiple scattering events within the material, while the specular reflection is a surface effect.

Since in transparent object's a certain amount of light passes through the object instead of getting scattered back, it is justified (and quite common) to decrease the amount of reflected light proportionally when adding transparency to a material.

For the specular reflection it is not obvious why a transparent object should have a less bright highlight. Think of a glass-like material. On the other hand in visualization transparency is often used to reduce the overall visibility of an object. In this case a fully transparent object would be expected to be completely invisible. There is no general rule to what the most correct or most useful settings are. Some results for different parameters will be shown at the end of this section.

Summary

The most important observations from the above equations are, that

(a) the transparency of a layer of absorbing material is proportional to the length of the way a light beam has to pass through the material on its way from a light source to the eye of the observer.

(b) This transparency is view dependent.

(c) The absorbed, emitted, and reflected light intensities vary as a function of transparency.

3.4.2 Implementation

In this subsection we describe how these results can be implemented with standard polygon rendering systems, like an OpenGL capable graphics board.

We also present a way to perform the view dependent calculations using the texture hardware. This makes it possible to achieve the same frame rates as with non-view dependent transparency.

Blending

Putting the equations from subsection 6.2.1 together, we obtain what often is called *blending*. Assume that the transparency of a surface at a particular point is θ and it's light intensities resulting from the *Phong*-lighting model are I_{diffuse} and I_{specular}. If the light intensity originating from objects behind this surface is $I_{\text{Background}}$, then the light that strikes the observers eye will be:

$$I_{\text{Result}} = \theta I_{\text{Background}} + (1-\theta)\left(\frac{q}{\kappa} + I_{\text{diffuse}}\right) + I_{\text{specular}} \tag{3.23}$$

or

$$I_{\text{Result}} = \theta I_{\text{Background}} + (1-\theta)\left(\frac{q}{\kappa} + I_{\text{diffuse}} + I_{\text{specular}}\right). \tag{3.24}$$

If θ is given, Equation (3.24) can be evaluated directly by the graphics hardware, since it is possible to blend an objects color (the expression in brackets) with the background, i.e. to compute a weighted average with weight θ.

In case of Equation (3.24) the highlight gets attenuated also. If this is not desired, Equation (3.23) should be used instead. In this case the highlight is added to the pixel after the blending

operation. This can either be implemented using a multi-pass technique, or it can be approximated by rewriting (3.24):

$$\begin{aligned} I_{\text{Result}} &= \theta I_{\text{Background}} + (1-\theta)\left(\frac{q}{\kappa} + I_{\text{diffuse}}\right) + I_{\text{specular}} \\ &= \theta I_{\text{Background}} + (1-\theta)\left(\frac{q}{\kappa} + I_{\text{diffuse}} + \frac{I_{\text{specular}}}{(1-\theta)}\right) \end{aligned} \tag{3.25}$$

Although the identity (3.25) is exact, clipping problems will occur especially if $(1-\theta)$ gets small. This is due to the fact that many OpenGL implementations only allow a limited range for I_{specular} that could be exceeded by $\frac{I_{\text{specular}}}{(1-\theta)}$.

Apart from this potential problem, the implementation of this view-dependent transparency model is straight forward. For each vertex equation (3.21) has to be evaluated, to compute the transparency value for the current view at that vertex. Furthermore it has to be assured, that polygons are drawn in a depth sorted way, as discussed for transparent lines in section 3.3.3. Due to the term $\boldsymbol{V} \cdot \boldsymbol{N}$ Equation (3.21) has to be reevaluated whenever the view direction changes, i.e. when the object is rotated.

Hardware Accelerated Implementation

To accelerate the repeated evaluation of a function for input parameters from a given interval, the function can be approximated by a lookup to a precomputed table of function values, potentially combined with a linear or higher order interpolation scheme. Though originally designed for a different purpose, texture hardware present in all modern graphics systems can be used to implement this lookup and a linear interpolation, similar to our method described in section 3.3:

When texture mapping is used, for each vertex of a polygonal model a vector of texture coordinates is specified. The first components of these vectors are used to lookup color or transparency values in a one-, two-, or three-dimensional texture map. The key feature for our application is the so called texture transformation matrix, which is applied in hardware to the texture coordinate vectors before the actual lookup is done.

These computations are performed with homogeneous coordinates. To compute the view-dependent part in (3.21) we initialize the texture transformation matrix with the components of the view direction $\boldsymbol{V}$ and use the surface normal as texture coordinate.

$$M = \frac{1}{2}\begin{pmatrix} \boldsymbol{V}_1 & 0 & 0 & 0 \\ \boldsymbol{V}_2 & 0 & 0 & 0 \\ \boldsymbol{V}_3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

The first component of the transformed homogeneous texture vector $\boldsymbol{t} = \boldsymbol{t}_0 M$ then evaluates to

$$t_1 = \frac{1}{2}(\boldsymbol{V} \cdot \boldsymbol{N} + 1). \Rightarrow \boldsymbol{V} \cdot \boldsymbol{N} = 2t_1 - 1$$

The fourth row of the matrix M has been chosen like this to assure that t_1 lies always in the range $0 \ldots 1$, so that it can be used as an index to a one-dimensional texture map. The texture map P then is initialized by evaluating (3.21). It is illustrated in Figure 3.13:

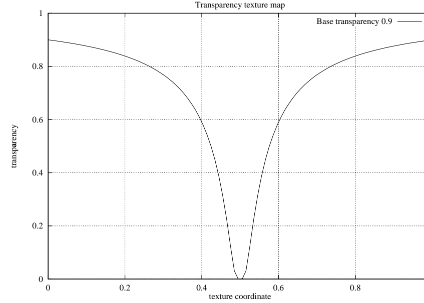

Figure 3.13: View dependent transparency for viewing angles -90 ... 90 degrees.

$$P(t_1) = \theta_0^{\frac{1}{|2t_1-1|}} \tag{3.26}$$

The texture map will be used to lookup the transparency for each fragment after the lighting computation has taken place and before it is drawn to the frame buffer.

This way, the result from a standard OpenGL lighting step can be accomplished with an appropriate view-dependent and locally varying transparency. Two problems that remain are (i) the highlight gets attenuated by the same amount as the diffuse color and (ii) it is not obvious how the view-dependent emissive color can be specified.

For objects with constant material properties an elegant way to solve both problems is to not only employ the texture hardware for the alpha computation but for the complete shading calculation.

With simple geometric considerations the reflection vector $\boldsymbol{R}$ in Equation (3.1) can be written as:

$$\begin{aligned} \boldsymbol{R} &= 2(\boldsymbol{L}\cdot\boldsymbol{N})\boldsymbol{N} - \boldsymbol{L} \\ \Rightarrow \boldsymbol{R}\cdot\boldsymbol{V} &= 2(\boldsymbol{L}\cdot\boldsymbol{N})(\boldsymbol{V}\cdot\boldsymbol{N}) - \boldsymbol{V}\cdot\boldsymbol{L} \end{aligned} \tag{3.27}$$

We see that the color per vertex according to Equation (3.1) is a function of the two dot products $\boldsymbol{L}\cdot\boldsymbol{N}$, $\boldsymbol{V}\cdot\boldsymbol{N}$, and values that are constant within the whole scene. Initializing the texture transformation matrix with the two vectors $\boldsymbol{L}$ and $\boldsymbol{V}$ this dot products can again be computed in hardware and be used as lookup to a two dimensional texture map:

$$M = \frac{1}{2}\begin{pmatrix} V_1 & L_1 & 0 & 0 \\ V_2 & L_2 & 0 & 0 \\ V_3 & L_3 & 0 & 0 \\ 1 & 1 & 0 & 2 \end{pmatrix}$$

The second component of the transformed texture coordinate becomes then

$$\begin{aligned} t_2 &= \frac{1}{2}(\boldsymbol{L}\cdot\boldsymbol{N} + 1) \\ \Leftrightarrow \boldsymbol{R}\cdot\boldsymbol{V} &= 1 - 8t_2 + 8t_2^2 \end{aligned} \tag{3.28}$$

If a 2D texture $P(t_1, t_2)$ is initialized with the Phong light model $I(t_1, t_2)$ then the result of the texture lookup can be directly used as final color value without any additional lighting computations.

Since the hardware will for each triangle first interpolate the texture coordinates (actually these are the normal vectors in our case) and then do the lookup, we get a real Phong shading, as opposed to Gouraud shading where the lighting calculation is performed per vertex and the resulting color is interpolated.

An important special case is a so called head-light, where the light is assumed to be always in view direction: $\boldsymbol{L} = \boldsymbol{V}$. In this case obviously a one dimensional texture is sufficient.

3.4.3 Results

In this section we have shown that transparency model provided by standard OpenGL yields unsatisfactory results. We have proposed an improved transparency model and shown how it can be implemented in hardware.

The per-view evaluation of the exponential function can be greatly accelerated by using precomputed tables. Texture hardware can be used to implement the computation of the table index and the lookup. This yields an additional performance increase of about 15% (measured on an SGI Onyx3400 with InfiniteReality3 graphics) to 30% (PentiumIII with Geforce2MX).

For constant colored objects the complete shading calculation including the view-dependent emission and a non-attenuated highlight can be implemented with the texture hardware. The OpenGL lighting can then be switched off and no normals have to be sent into the pipeline. This yields another 30%-40% of performance increase and very appealing images.

Figure 3.14 illustrates these results. It is obvious how much shape information is added by the view dependent transparency compared to the standard case.

Even multiple nested objects can be perceived when the transparency is near 1 and still the shape of the objects is visible. If the depth relation of the objects cannot be recognized directly from the still image it will immediately be seen with an interactive rotation.

As we have pointed out earlier, interactive exploration of three dimensional scenes can greatly increase the ease of perception compared to looking at still images. This is especially true for the effects described in this section. Even the shape of a fully transparent object with a highlight as the only evidence of existence can be perceived quite clearly when changing the view.

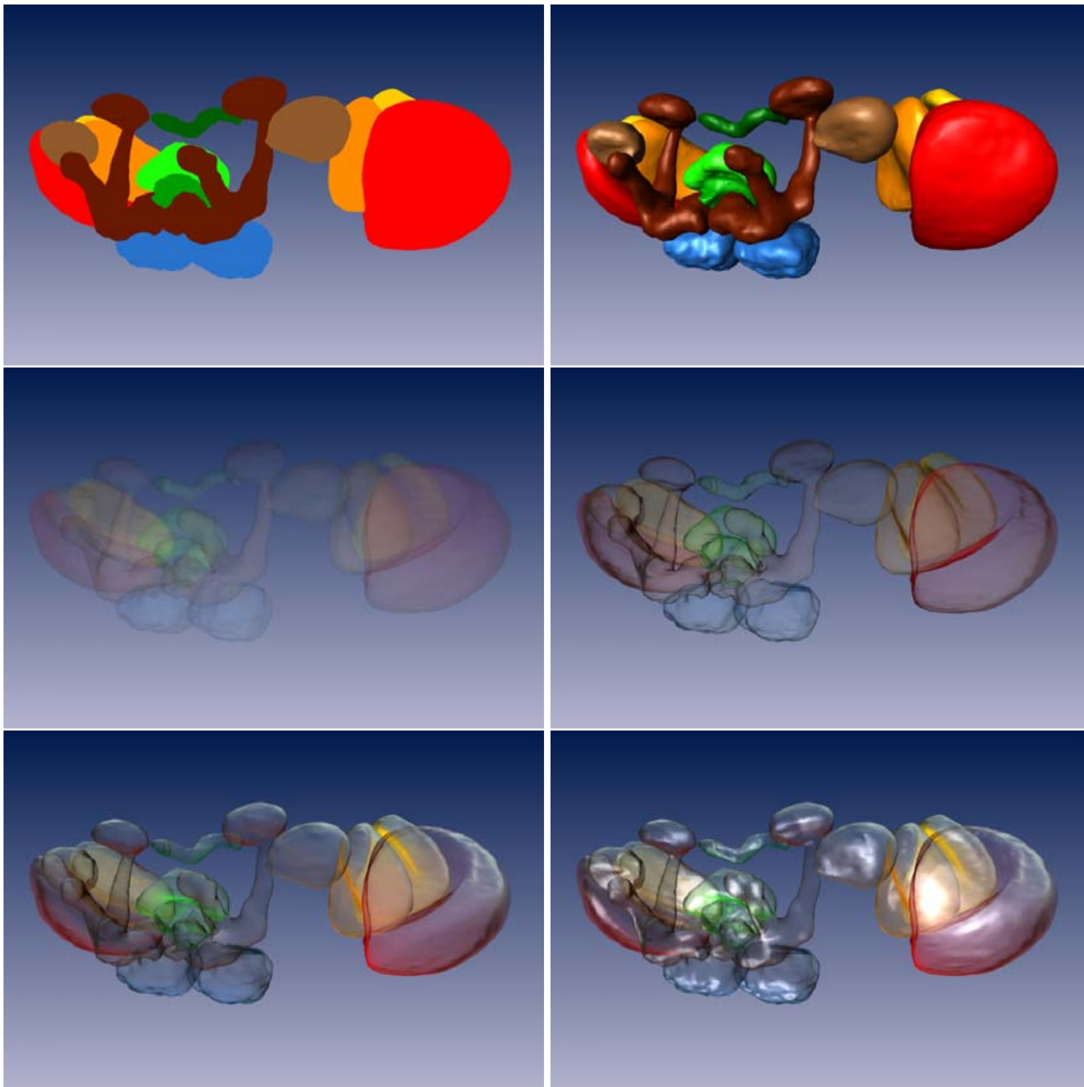

Figure 3.14: Shading is important to understand the shape of three dimensional objects as can be seen by comparing image (i), which is not shaded and image (ii). If a high constant transparency is used, the shaded intensity gets attenuated so that the visual shape information is significantly reduced again (iii). This problem can be tackled by adjusting the transparency according to the apparent thickness of a layer (iv). Adding unattenuated specular reflection can further increase understanding (v,vi).