Notes on Scientific Visualization

Paulo Eduardo Rauber

2015

1 Introduction

The purpose of data visualization is to enable *insight* about some phenomenon. Visualizations are useful for both hypothesis creation and evaluation. Often, pictorial representations are the best way to summarize information in a compact and understandable way. A very simple and familiar example is a histogram, a graphical depiction of a data distribution.

A visualization that displays a well defined quantity is called quantitative. A visualization that displays data for the purposes of exploration is called qualitative.

The discipline of data visualization can be further divided into three subfields: scientific visualization, information visualization and visual analytics. Scientific visualization is concerned with the representation of phenomena that are naturally described as a function of space and, optionally, time. Information visualization is concerned with the representation of abstract objects such as functions and graphs. Visual analytics is the combination of information visualization and interactive interfacing.

2 Rendering

This section describes the basics of rendering three dimensional scenes. In this context, rendering is the process that transforms a three dimensional scene into a two dimensional image.

A *rendering equation* describes, for every point in the scene, the relation between incoming light, outgoing light and the properties of the material in that point.

Rendering equations used in practice are simplifications of the actual physical phenomenon they model. Two examples are the radiosity methods, which are good at producing soft shadows, and ray tracing methods, which are good at simulating shiny surfaces. These methods are examples of global illumination, because the illumination of a given point is dependent on many other points in the scene.

Local illumination methods model the rendering equation independently for every point. One such method is called Phong lighting.

In Phong lighting, the illumination at a point p in the surface of an object in the scene is given by

$$I(p, v, L) = c_{\text{amb}} + I_l(c_{\text{diff}} \max(-L \cdot n, 0) + c_{\text{spec}} \max(r \cdot v, 0)^{\alpha}).$$

In this equation, p is the position vector of the surface point, n is the unit vector normal to the surface at that point, v is an unit vector pointing to the viewpoint, I_l is the intensity of the light, L is an unit vector in the direction of the light, and $r = L - 2(L \cdot n)n$ is the direction in which the light is reflected by the surface at point p. The constants c_{amb}, c_{spec} and c_{diff} are called ambient, specular and diffuse. The ambient constant controls the illumination due to to indirect illumination (since that is not present in this lighting model). The diffuse constant controls the output lighting in proportion to how well the light hits the surface against the normal. High values for the diffuse constant are used to simulate matte surfaces. The specular constant controls the output lighting in proportion to how well the light is reflected in the direction of the viewpoint. High values for the specular constant are used to simulate glossy surfaces.

Rendering colored scenes using the Phong lighting model is accomplished by having one distinct illumination equation for each color channel, each one with distinct material constants.

Naturally, the Phong illumination equation cannot be computed for a continuous set of points. However, a three dimensional surface can be approximated or interpolated from a grid composed of polygons. These polygons may be, for instance, *quads*. A quad is a 4-tuple of vectors (called vertices) in $3D$ that represents a surface (polygon).

In *flat shading*, the illumination for any point p in a surface $\mathcal{S}$ represented by a set of quads is given by the illumination of the point p_c, the center of the quad to which p belongs, as given by the Phong lighting model. Flat shading leads to faceted surfaces.

In *smooth shading*, particularly the technique called Gouraud shading, can be used eliminate the faceted surfaces given by flat shading. In Gouraud shading, the illumination of each vertex in the quad is computed by the Phong lighting model. All the points in the quad are associated to an illumination by interpolation from the illumination of each vertex in the quad.

If the objective is to render a continuous surface $\mathcal{S} = \{(x, y, f(x, y))|(x, y) \in \mathbb{R}^2\}$, a vector normal to (x, y) is $n(x, y) = (-\frac{\partial f(x,y)}{\partial x}, -\frac{\partial f(x,y)}{\partial y}, 1)$. If the objective is to render a surface that is represented by a grid, the normal at vertex v in a quad can be defined as the average of the vectors that are normal to the center of every quad that has v as a vertex.

Textures can be added to surfaces by defining a texture coordinate (s, t), $0 \leq s, t \leq 1$, to each vertex in a quad. The texture image is a function $f : [0, 1] \times [0, 1] \to [0, 1]$. The polygon in the texture image defined by the quad's vertices texture coordinates is warped to fit in the quad. The texture values are thus defined for every point in a quad, and can be used, for example, to multiply their illumination values before presentation.

A virtual camera is necessary to define the rendered image given the three dimensional scene and the lights. A virtual camera, that uses perspective projection, is defined by extrinsic and intrinsic parameters. The extrinsic parameters are the camera location vector e, a location pointed at by the camera c and a vector u, called the up vector, that is orthogonal to $c - e$ and indicates how the camera is rotated with respect to the viewing direction. The intrinsic parameters are the focal length, field of view and aspect ratio of the view area. A plane spanned by u and a vector orthogonal to both $c - e$ and u, at a distance called focal length from the position of the camera in the direction $c - e$, is called view plane. A vector orthogonal to both $c - e$ and u can be obtained by the cross product $(c - e) \times u$. The view area is the rectangle in the view plane whose center is on the line defined by view direction $(c - e)$ and whose vertical axis is parallel to u, with size defined uniquely by the field of view and aspect ratio. The aspect ratio gives the ratio of the horizontal to the vertical size of the view area. The field of view is the angle formed between the location vector e and the top and bottom locations of the view area (along its axis parallel to u). Objects between the camera and the view plane (in the direction $c - e$) are hidden by the projection. It is also possible to define z_{far} , a distance (in the viewing direction) after which the camera sees no objects. The focal length z_{near} and z_{far} define a pyramid frustum on the scene containing objects that could be seen in the rendered image.

Given the scene, lights and virtual camera, the next step is to define a viewport transform. This transform defines how the pixels on the screen relate to coordinates in the view area. The view area is usually warped to the viewport (a two dimensional rectangle on the screen), using a linear transform. The aspect ratio of the view area is usually maintained by fitting it in the best possible way in the viewport.

When all these objects are defined, rendering becomes the problem of finding the value of each pixel in the viewport by mapping it to a point in the view area, finding a point in a surface that is mapped to the point in the view area, computing its illumination, and mapping textures to it. It is important to note that this naive description is not how this pipeline is implemented in practice.

Transparency is obtained through blending. Blending is a technique in which each object in a scene is rendered sequentially onto a *frame buffer*. The intensity value for a pixel in the frame buffer after an object is drawn is then set as a linear combination of the previous value for that pixel and the value computed by the rendering of that pixel.

3 Data Representation

Two kinds of quantities need to be represented by visualization techniques: continuous and discrete. Continuous data often needs to be sampled before being stored in a computer, and its visualization is often studied in the field of scientific visualization. Discrete data visualization is most commonly studied in the field of information visualization.

Continuous data can be modeled as a function $f : D \to C$, where $D \subseteq \mathbb{R}^d$ and $C \subseteq \mathbb{R}^c$. In this context, f is called a d-dimensional, c-valued function.

A function is continuous if, for every $p \in C$,

$$\forall \epsilon > 0, \exists \delta > 0, \text{ such that, if } 0 < ||x - p|| < \delta, x \in C, \text{ then } ||f(x) - f(p)|| < \epsilon. \tag{1}$$

Intuitively, a function f is continuous if, for every point p in its domain, for any non-empty interval I_C around $f(p)$, there is another non-empty interval I_D around p such that every point in I_D is mapped to a point in I_C. In an informal sense, this guarantees that small changes in p lead to small changes in $f(p)$.

A function f belongs to $\mathcal{C}^k$ if an only if it is continuous and all of its derivatives up to order k are continuous. If the derivatives of f of any order are continuous, $f \in \mathcal{C}^\infty$.

A data set is defined by a triplet $\mathcal{D} = (D, C, f)$, where $D \subseteq \mathbb{R}^d$, $C \subseteq \mathbb{R}^c$ and $f : D \to C$. The natural number d is also called geometrical dimension. The dimension $s \leq d$ of the manifold in which the data is embedded is called topological dimension. This concept will not be formalized in this text. It can be intuitively understood as the number of independent variables needed to identify an element of the domain.

This text will be concerned with datasets where $d = 3$ and $c \in \{1, 2, 3, 6\}$. It will also be concerned with $s = \{1, 2, 3\}$, which correspond, respectively, to curves, surfaces and volumes. The topological dimension is also called the data set dimension. The codimension of $\mathcal{D}$ is defined as $d - s$.

Sampling continuous data is essential for analysis and rendering. Given the sampled data, it should be possible to reconstruct an approximation for the original, continuous data. Interpolation is the technique used to obtain function values from function sample points.

A sampled data set should be accurate (able to approximate the original data set), minimal (containing the least amount of samples), generic (useful for processing), efficient (reconstruction should be fast) and simple (understandable).

Let $\mathcal{D} = (D, C, f)$ be a continuous data set. Consider the set of sample points $P = \{(p_i, f_i) | 1 \leq i \leq N\}$, where $(p_i, f_i) = (p_i, f(p_i))$. The goal is to define a reconstructed function $\tilde{f}$ that approximates the original function f. This can be accomplished as follows:

$$\tilde{f}(x) = \sum_{i=1}^{N} f_i \phi_i(x).$$

The function $\phi_i(x) : D \to C$ is called a basis function. In simple terms, the reconstruction is defined as an averaging of sampled function values, weighted by the basis functions.

Let p_i be any sampled point ($(p_i, f_i) \in P$). If $\phi_i(p_i) = 1$ and $\phi_i(p_j) = 0$ for any other sampled point p_j, then the basis function ϕ_i is orthogonal. If $\sum_{i=1}^{N} \phi_i(x) = 1$ for every $x \in D$, the basis functions are normal.

A grid, or mesh, is a partition of the domain D into sets called cells. Since the grid is a partition, $\cup_i c_i = D$ and $c_i \cap c_j = \emptyset$, for $i \neq j$. The most commonly used cell types are lines (for $s = 1$), polygons (for $s = 2$) and polyhedra (for $s = 3$).

When a grid is defined for a domain D and a set of samples $P = \{(p_i, f_i) | 1 \leq i \leq N\}$ such that each p_i is the center of a cell c_i, the constant basis functions ϕ_i^0 (for $1 \leq i \leq N$) can be defined as

$$\phi_i^0(x) = \begin{cases} 1 & \text{if } x \in c_i, \\ 0 & \text{if } x \notin c_i. \end{cases}$$

These functions are orthogonal and normal. Intuitively, a point $x \in c_i$ is mapped to the same value as the sample point p_i. Constant basis functions are simple to implement and computationally inexpensive for any cell shape. However, they give a poor, piecewise-constant, reconstruction.

Linear basis functions are an alternative to constant basis functions. In this case, however, it is necessary to take the cell type into account. The use of (bi)linear basis functions will be illustrated using quadrilateral polygons (quads) as an example. Our objective is to approximate the function $f : [0, 1] \times [0, 1] \to \mathbb{R}^n$ with the function $\tilde{f}$, defined using only basis functions and sample points.

Consider the quadrilateral $(v_1, v_2, v_3, v_4) = ((0, 0), (1, 0), (1, 1), (0, 1))$, which will be called the reference cell. The first step is to define basis functions to interpolate within the reference cell. For a quad, let

$$\begin{aligned} \Phi_1^1(r, s) &= (1 - r)(1 - s), \\ \Phi_2^1(r, s) &= r(1 - s), \\ \Phi_3^1(r, s) &= rs, \\ \Phi_4^1(r, s) &= (1 - r)s. \end{aligned}$$

The reconstruction within the reference cell can be written as

$$\tilde{f}(r, s) = \sum_{i=1}^{4} f_i \Phi_i^1(r, s),$$

where $f_i = f(v_i)$. The basis functions Φ_i^1 are orthogonal and normal. Intuitively, the value for a point (r, s) is an average of the sample values for the vertices of the reference cell, weighted by the distance to each vertex.

The reconstruction defined in the reference cell can be used on a grid composed of such cells by defining a transformation $T : [0, 1] \times [0, 1] \to \mathbb{R}^3$ from reference cell coordinates to domain coordinates.

For quads, the transformation T is defined as

$$(x, y, z) = T(r, s) = \sum_{i=1}^{4} p_i \Phi_i^1(r, s),$$

where p_i is the point in the domain coordinates corresponding to v_i. Notice that if $(r, s) = v_i$, $(x, y, z) = p_i$. The transformation T can often be made invertible by restricting its codomain to a particular cell in the original grid.

Consider again the goal of reconstructing a continuous data set $\mathcal{D} = (D, C, f)$ from the set of sample points $P = \{(p_i, f_i) | 1 \leq i \leq N\}$, where $(p_i, f_i) = (p_i, f(p_i))$ and $C(p_i)$ is the set of cells that contain the point p_i. Consider that $\mathcal{D}$ has topological dimension 2 and the reference cells are quads.

We define $\tilde{f}$ as

$$\tilde{f}(x, y, z) = \sum_{i=1}^{N} f_i \phi_i^1(x, y, z),$$

where

$$\phi_i^1(x, y, z) = \begin{cases} 0 & \text{if } (x, y, z) \notin c,\ \forall c (c \in C(p_i)), \\ \Phi_j^1(T^{-1}(x, y, z)) & \text{if } (x, y, z) \in c,\ c \in C(p_i),\ c = (v_1, v_2, v_3, v_4),\ v_j \text{ corresponds to } p_i. \end{cases}$$

The linear basis functions are once again orthogonal and normal. Also, $\tilde{f} \in \mathcal{C}^0$.

The discussion above is limited to quads, limiting the grids to datasets with topological dimension 2. However, the same idea can be extended for several types of cells.

In summary, given a grid partitioned into cells, values sampled at the known cell vertices (or centers) and a set of reference basis functions, it is possible to reconstruct a function. The cell shapes, together with the basis functions, determine different cell types. The number and position of sample points determine the different grid types.

The dimensionality d of the cells has to be the same as the topological dimension of the sampled domain D. For example, if domain D is contained in a curve, the cells should be lines. If it is contained in a surface, the cells should be polygons. If it is contained in a volume, the cells should be volumetric cells, such as tetrahedra.

The following details cell types, together with their reference coordinates, linear basis functions and inverse transformation to the domain coordinates.

A vertex cell is the simplest cell type and has dimension 0. The vertex cell has the reference basis function $\Phi_1^0 = 1$. The inverse transformation from domain coordinates to reference coordinates for a point p which belongs to the cell is simply $T_{\text{lin}}^{-1}(p) = v_1$.

A line cell has dimension 1 and its reference coordinates are $c = (v_1, v_2) = (0, 1)$. Its reference linear basis functions are $\Phi_1^1(r) = 1 - r$ and $\Phi_2^1(r) = r$. For a point p contained in a line cell from p_1 to p_2, the inverse transformation is defined as

$$T_{\text{lin}}^{-1}(x, y, z) = T_{\text{lin}}^{-1}(p) = \frac{||p - p_1||}{||p_2 - p_1||},$$

which intuitively corresponds to the ratio between the length of the segment from p_1 to p to the total length of the line.

The triangle cell has dimension 2 and is represented by the reference coordinates $c = ((0, 0), (1, 0), (0, 1))$. Its reference linear basis functions are given by

$$\Phi_1^1(r, s) = 1 - r - s,$$
$$\Phi_2^1(r, s) = r,$$
$$\Phi_3^1(r, s) = s.$$

The inverse transformation from domain coordinates to reference coordinates for a point p which belongs to the cell is defined as

$$T_{\text{tri}}^{-1}(x, y, z) = T_{\text{tri}}^{-1}(p) = \left(\frac{||(p - p_1) \times (p_3 - p_1)||}{||(p_2 - p_1) \times (p_3 - p_1)||}, \frac{||(p - p_1) \times (p_2 - p_1)||}{||(p_3 - p_1) \times (p_2 - p_1)||} \right).$$

This transformation can be understood intuitively as a conversion from domain coordinates to barycentric coordinates.

The quad cell has dimension 2 and is represented by the reference coordinates $c = ((0, 0), (1, 0), (1, 1), (0, 1))$. Its reference *bilinear* basis functions are given by

$$\Phi_1^1(r, s) = (1 - r)(1 - s),$$
$$\Phi_2^1(r, s) = r(1 - s),$$
$$\Phi_3^1(r, s) = rs,$$
$$\Phi_4^1(r, s) = (1 - r)s.$$

The inverse transformation from domain coordinates to reference coordinates cannot be computed analytically (unless the quad is a rectangle), but can be computed in other ways, which are omitted. Notice that the basis functions are not linear on r and s. They are said to be bilinear.

Flat shading is implemented by constant basis functions, while smooth (Gouraud) shading is implemented by (bi)linear basis functions. The geometry of a surface can also be interpolated by constant basis functions (staircase approximation) or (bi)linear basis functions (polygonal approximation). It is not possible to combine constant geometry with (bi)linear lighting, since constant geometry is discontinuous in general.

The tetrahedron cell has dimension 3. Its reference coordinates are $c = ((0,0,0),(1,0,0),(0,1,0),(0,0,1))$. Its reference trilinear basis functions are given by

$$\begin{aligned}\Phi_1^1(r,s,t) &= 1-r-s-t,\\ \Phi_2^1(r,s,t) &= r,\\ \Phi_3^1(r,s,t) &= s,\\ \Phi_4^1(r,s,t) &= t.\end{aligned}$$

Its inverse transformation can be consulted in [1]. The idea is similar to the barycentric coordinates used with triangular cells.

The hexahedron cell is another 3 dimensional cell. It is defined by eight reference coordinates, in a similar way to a quad cell. Its reference basis functions can be consulted in [1], and the inverse transformation is also not analytical (unless the hexahedron is a rectangle).

Some applications also use quadratic cells (not to be confused with quad cells), which are interpolated using quadratic basis functions. In this case, additional points in the edges and center of the cells are needed in comparison to linear interpolation. The surfaces interpolated from grids of quadratic cells are in $\mathcal{C}^1$.

The main reason for using these cells is the ease of defining reference linear basis functions and inverse transformations from domain coordinates to the reference cell.

The following is a summary of the most important grid types in visualization.

An uniform grid is an axis-aligned *box* of a given geometrical dimension d. If $d = 1$, the grid is composed of line segments. If $d = 2$, the grid is composed of rectangles. If $d = 3$, the grid is composed of parallelepipeds. Points are equally sampled along each axis (standard basis vectors) of D.

For each dimension d_i, it is necessary to store (m_i, M_i, N_i) that represent, respectively, the minimum value along that dimension, the maximum value along that dimension and the number of sample points along that dimension. Each sample point can be uniquely identified by d coordinates $(n_1, \ldots, n_d)$. The coordinates of a point p_i can be written as $(m_1 + n_1\delta_1, \ldots, m_d + n_d\delta_d)$, where $\delta_i = \frac{M_i - m_i}{N_i - 1}$ is the space between samples. These coordinates also can be combined into one single identifier i:

$$i = n_1 + \sum_{k=2}^{d} \left(n_k \prod_{l=1}^{k-1} N_l \right).$$

Storing a d dimensional grid requires $3d$ floating point numbers for specifying the grid and $\prod_{i=1}^{d} N_i$ values for each sampled point.

It is often more efficient to sample a data set non-uniformly. Rectilinear grids are also axis-aligned and every point can be represented by a single identifier i. However, each dimension d_i is sampled by pre-defined steps $\delta_{i,j}$, $0 \leq j < N_i$. Therefore, a sample point defined by grid coordinates $(n_1, \ldots, n_d)$ corresponds to the point $p_i = (x_1, \ldots, x_d)$ in domain coordinates, where $x_i = m_i + \sum_{j=0}^{n_i - 1} \delta_{ij}$. The cells in a rectilinear grid may be lines, rectangles or parallelepipeds.

Structured grids allow the explicit positioning of every sample point p_i in domain coordinates $(x_{i1}, \ldots, x_{id})$. In this case, it is necessary to store d values for each point together with their N sample values. In a structured grid, the cells are defined implicitly by the order of points. For instance, each sequence of three points in domain coordinates may represent a triangle.

Unstructured grids are the most general kind of grid. An unstructured grid is represented by a set of sample points $\{p_i\}$, for $1 \leq i \leq N$, and a set of cells $\{c_j\}$. Each cell c_j is a tuple $(v_1, \ldots, v_K)$ such that $v_k \in \{1, \ldots, N\}$ is the index of a sample point. If cells share a sample point, this may also be explicitly represented. This is useful for computations involving the vertices of cells. For a grid of C d-dimensional cells with V vertices per cell and N sample points, $CV + dN$ storage space would be required.

The representation of the codomain C of a data set $\mathcal{D} = (D, C, f)$ is also important. The sample values $f_i = f(p_i)$ of the function f sampled at points p_i belong to $C \subseteq \mathbb{R}^c$ and are called attribute data. If $c = 1$, the attribute data

is scalar, and each sample point is associated to a real number. Scalar attributes may represent several physical quantities, such as temperature, pressure, density etc. If $c = 2$ or $c = 3$, the attribute data is vectorial, and each sample point is associated to a vector. These vectors may represent positions, directions, force, gradients etc.

Color attributes are a particular case of vector attributes, usually 3-dimensional, and represent colors in a predefined color space. The RGB colorspace is an additive colorspace, where each component of a color $c = (R, G, B)$ corresponds to the amount of red, green and blue, respectively, ranging from 0 to 1. When $R = G = B$, the color is a shade of gray. The RGB colorspace can be represented by a cube. The diagonal between $(0, 0, 0)$ and $(1, 1, 1)$ corresponds to gray values, and all representable colors are in the cube.

Another important colorspace is HSV. In HSV, each color c is represented by a coordinate (H, S, V), where H is the hue, S is the saturation and V is the value. The hue distinguishes between colors of different wavelengths, the saturation represents the purity of color with respect to white and the value is related to brightness. The HSV system can be represented by a cylinder. Each circular slice in the cylinder (perpendicular to its main axis) corresponds to a value V. The distance from the center to the edge of the circle represents a saturation, and the hue is the angle between a given point and a pre-defined axis divided by 2π. When converting from RGB to HSV, it is impossible to represent the hue for a grayscale value uniquely.

There are many other color spaces, which are useful for different purposes and are outside the scope of this text.

Cell to vertex resampling allows the computation of values for a vertex given the cells to which it belongs. One example is computing the normal at a vertex using the adjacent cells. In vertex to cell resampling, the objective is to compute the value of a cell from the value of its vertices. Computing the cell normal from the vertex normals is an example.

Subsampling and supersampling are techniques that create datasets with less or more sample points from a given sampled data set.

Subsampling is useful for optimizing efficiency requirements of implementations. In uniform subsampling, which can be performed in uniform, rectilinear and structured grids, every k-th point along a dimension is kept and the rest are discarded.

Supersampling is useful to add detail below the sampling frequency. The counterpart to uniform subsampling is uniform supersampling, which introduces k into every cell of the original data set.

Given a set of sampled points and values without a grid, it is necessary to use scattered point interpolation techniques. There are two important options to interpolate scattered points: constructing a grid using triangulation or performing gridless interpolation.

4 Visualization Pipeline

The visualization pipeline has four stages: importing, filtering, mapping and rendering. This pipeline can be seen as a function from the set all possible raw data to the set of all possible images. The objective is to create images that allow insight about the data. Insight can be seen as a function from the set of all possible images to the set of all possible raw data. In this analogy, insight is the inverse of the visualization.

Importing data consists in finding a representation for the data as a data set, as defined in the previous section. It involves dealing with the raw representation of the data in a storage device or database, as well as the sampling of the data.

Filtering data consists in choosing a subset of the data corresponding to the features of interest, which includes restricting the domain and codomain of the data set.

Mapping consists in transforming the data set into a representation in terms of visual features. These visual features include all design choices such as position, size, color, texture, shading and motion. Mapping encodes the decisions about what the visualization shows and how it is shown. It is the most crucial step in data visualization. Ideally, the mapping step should be injective, meaning that it should be possible to imagine the (filtered) data from the scene represented by the mapping. The mapping can use explicit or implicit conventions to allow insight from the resulting scene.

It is very important to not allow deviations that would lead users to incorrect conclusions from what they see.

Rendering is the step responsible for using computer graphics to generate an image from the scene specified in the mapping step.

5 Scalar visualization

A scalar data set is a representation of a function $f : D \to \mathbb{R}$, where $D \subseteq \mathbb{R}^d$ and $2 \leq d \leq 3$.

Color mapping is the most commonly used scalar visualization technique. A color is associated to every point $x \in D$ based on the value $f(x)$. This is usually done using a function $c : \mathbb{R} \to \mathcal{C}$, where $\mathcal{C}$ is a set of colors, represented analytically (transfer function) or a table (colormap).

A colormap may be considered effective when, from the user's perspective, it is possible to infer the original scalar value of a point by looking at its color.

Colormaps may give information about absolute scalar values, scalar value ordering, scalar value difference, scalar values in particular range and rates of change. A color legend is essential in all of these cases.

There are several widely used classes of colormaps: rainbow, grayscale, two-hues, heat and diverging. Each has perceptual weaknesses and strengths, and are adequate in different scenarios. In particular, isoluminant (constant luminance) colormaps are necessary when the scene is shaded, to avoid misrepresentations.

The zebra colormap is slightly different than these, and is designed to convey information about rates of change (including direction). The range of values is discretized into a number of steps, whose associated color alternates between black and white. The width of the strips observed in the image convey information about variation in the scalar field. Thin strips represent regions with high variation and long strips represent regions of low variation.

The color mapping function also needs to be injective from the perspective of the user.

There are two alternatives to associate a color to every point on a surface defined by a grid of cells:

$$C'(p) = \sum_i c(f_i)\phi_i(p),$$

or

$$C(p) = c\left(\sum_i f_i\phi_i(p)\right),$$

where p is a point on the surface, $f_i = f(p_i)$ are the scalar sample values for the vertices p_i of the cell that contains p, and ϕ_i are the associated basis functions. The second alternative is always superior, since the first does not guarantee that the interpolated color is even present in the original colormap. In practice, the texturing mechanism can be used to implement the second alternative, while the first is implemented associating the color directly to each vertex in the surface. By defining a one dimensional texture representing the colormap and associating one texture coordinate to each vertex, the texture coordinates (normalized f_i) are interpolated and the final value is obtained directly from the texture map.

Color banding is the discretization of the colormap into a pre-defined number of colors. It can be used, for example, to visualize clearly the boundary between intervals of colors.

A data set is categorical if it represents a function from $f : D \to \mathbb{N}$ and each scalar value represents a distinct category of interest. In this case, only absolute scalar values, scalar value ordering (if defined) and scalar values in a particular range are usually of interest.

Additional issues need to be considered when designing colormap, including perceptual difficulties caused by color neighborhoods composed of perceptually similar colors, colorblindness, medium used to convey the images, difficulty to differentiate between high (low) luminance colors, and others.

Consider the function $f : D \to \mathbb{R}$. A contour line for the value x is the set $C(x) = \{p \in D | f(p) = x\}$. Contour lines are useful to visualize sets of points in the domain that have the same associated value. Consider a function whose domain $D \subseteq \mathbb{R}^2$. If contour lines for a sequence $x+\delta, x+2\delta, \ldots, x+n\delta$ are drawn on a plane representing the function's domain, for a constant δ, it is possible to infer the rate of variation in different directions. This technique is widely used in cartography to represent terrain inclination. In this case, in Cartesian coordinates, the contour $C(v)$ can also be interpreted as the intersection between the functions $f(x, y) = z$ and $z = v$ (plane at height v).

The following properties of isolines of a continuous function f are notable: they are closed (unless they reach the boundaries of the domain), they don't intersect themselves or each other and, at any point, are perpendicular to the gradient vector at that point.

Computing the contour $C(v)$ for a data set discretized into a grid of two dimensional cells, assuming piecewise linear reconstruction, can be performed in the following way. Consider a cell c in the grid, and let $e = (p_i, p_j)$ be the edge connecting vertex p_i to vertex p_j. If $\min(f_i, f_j) < v < \max(f_i, f_j)$, where $f_i = f(p_i)$ and $f_j = f(p_j)$, the edge e contains a point q in the contour, which is given by

$$q = \frac{p_i(f_j - v) + p_j(v - f_i)}{f_j - f_i}.$$

This is simply an average of p_i and p_j weighted by distance between the value of these vertices and v. Each edge can have either zero or one such q. By the equation above, this naive technique cannot deal with cases where $f_j = f_i$. We ignore such cases in what follows, assuming that there is no edge between vertices with the same associated scalar value. Each cell may have zero, two or four intersections. If a cell only has one intersection, it can be safely ignored, since either the same point belongs to another cell which has more than one intersection or the point is single. Exactly three intersections is impossible by the transitiveness of the $<$ relation. There is ambiguity in how to connect the intersection points for a cell when there are four intersections, which cannot be resolved when only a discrete data set is considered.

Another technique for computing the contour $C(v)$ for a data set discretized into a grid of two dimensional quadrilateral cells is called marching squares. In the marching squares algorithm, each vertex v_i in a cell receives a binary code representing whether $v_i < v$. Each (quadrilateral) cell can be represented by a code composed of four bits. This code is called a topological state. Each of these states (2^4 in total) is treated by highly optimized code, which computes the intersection points q and the unstructured polyline(s) defined by the contour. There is a single ambiguous case when symmetries are considered.

The marching cubes algorithm is a generalization of the marching squares algorithms for computing contours (also called isosurfaces) in a data set discretized into a grid of three dimensional hexahedron. In this case, there are 2^8 possible topological states, which can be reduced down to 15 by considering symmetries. Of these cases, 6 are ambiguous. These ambiguities are solved if tetrahedra are considered instead (marching tetraedra), or by enforcing coherence between neighboring cells. The marching cubes algorithm generates an unstructured grid (usually composed of triangles). However, it is still necessary to merge the endpoints of the triangles coming form every cell and computing the surface normals to render resulting contour.

The surfaces resulting from marching cubes need to be interpreted carefully, because artifacts may be introduced due to the subsampling of the original function. Using transparency, it is common to render several isosurfaces corresponding to different values in the same image.

The same result can be obtained by computing a contour in a slice of a three dimensional data set or by slicing an isosurface of the same data set. This leads to another way of computing isosurfaces, by integrating contours from different slices.

Consider a surface $D_s \subset \mathbb{R}^3$ with topological dimension $s = 2$ and the function $f : D_s \to \mathbb{R}$. A height plot of D_s is a surface $\mathcal{S} = \{x + f(x)n(x) | x \in D_s\}$, where $n(x)$ is the normal vector at point x in the surface D_s. Intuitively, the surface $\mathcal{S}$ is a deformed version of D_s where each point x corresponds to a point in D_s moved outwards in proportion to the value of $f(x)$. Height plots are usually easy to interpret and can be combined with color mapping. However, they may introduce problems such as occlusion.

6 Vector Visualization

A vector data set is a discrete representation of a vector *field* $f : D \to \mathbb{R}^3$.

Divergence and vorticity are scalar quantities that can be associated to every point in a vector field and visualized with the techniques presented in the previous section.

Let $f : D \to \mathbb{R}^3$, such that $f(x, y, z) = (f_x(x, y, z), f_y(x, y, z), f_z(x, y, z))$. The divergence $\operatorname{div} f : D \to \mathbb{R}$ of f is defined as

$$\operatorname{div} f(x, y, z) = \frac{\partial f_x(x, y, z)}{\partial x} + \frac{\partial f_y(x, y, z)}{\partial y} + \frac{\partial f_z(x, y, z)}{\partial z}.$$

Intuitively, if f is a field representing the velocity of a fluid, a positive divergence at point p can be understood as source that expels fluid outwards, a negative divergence can be understood as a sink that attracts fluid, and a divergence of zero represents that fluid is neither expanded nor compressed at that point.

The curl (rotor) $\operatorname{curl} f : D \to \mathbb{R}^3$ of a vector field $f : D \to \mathbb{R}^3$ is defined as

$$\operatorname{curl} f(x, y, z) = \left(\frac{\partial f_z(x, y, z)}{\partial y} - \frac{\partial f_y(x, y, z)}{\partial z}, \frac{\partial f_x(x, y, z)}{\partial z} - \frac{\partial f_z(x, y, z)}{\partial x}, \frac{\partial f_y(x, y, z)}{\partial x} - \frac{\partial f_x(x, y, z)}{\partial y} \right).$$

Consider the magnitude of the curl field f at point (x, y, z), $||\operatorname{curl} f(x, y, z)||$. Intuitively, high values for curl magnitude at point p represent that there is a vortex around p. It can be shown that $\operatorname{div} \operatorname{curl} f(x, y, z) = 0$ for any vector field f for every $(x, y, z) \in D$.

Vector glyphs are the most commonly used technique for visualizing vector fields. A vector glyph is an icon associated to every sample point of the vector data set, which encodes information such as direction, orientation and magnitude of the field.

Choosing the glyph for a given vector field is usually a trade off between how many glyphs can be displayed in a given area and the number of attributes that a glyph can convey.

One of the simplest glyphs is a line segment. The line glyph for a sample point x is represented by the pair $(x, x + kf(x))$, where k is a scaling constant that needs to be chosen according to the cell dimensions. Alternative options are cones, arrows, triangles etc.

In general, humans have difficulty to interpolate (mentally) the value of a vector field at a particular point from the surrounding vector glyphs. This contrasts with scalar fields, that are always displayed after being already interpolated for all pixels.

Glyphs can be placed randomly in the field's domain to avoid visual interference from the sampling pattern.

Occlusion is a big issue for vector fields with topological dimension equal to three. This can be alleviated by drawing semi-transparent glyphs. If the glyphs are monochromatic and scaled by the field's magnitude, it may become easy to spot regions of high magnitude just because of this difference in glyph length.

In the case of three dimensional data, it is also possible to find an isosurface for a given value of vector field magnitude and show glyphs only for points that lie on the isosurface. An isosurface for a given magnitude of a vector field f is said to be a stream surface if all vectors lying on the surface are tangent to it.

Any glyph can also be color colored to convey information about a scalar field.

Colors may also be used to represent orientation in two dimensional vector fields instead of glyphs. The HSV color space is ideal for this task: the angle of a vector gives the hue in the HSV color wheel (for a fixed saturation), and the vector magnitude may be represented by the color value. The main advantage of this representation is that the vector field does not need to be undersampled. However, the resulting image is often harder to interpret than a representation based on glyphs.

Direction in three dimensional vector fields can also be represented by color coding. Assuming that $||f(p)|| = 1$, one way to perform this coding is to define the RGB color of a point p to be $(|f(p) \cdot x|, |f(p) \cdot y|, |f(p) \cdot z|)$, where x, y and z are the usual Cartesian unit vectors.

Another possibility is to represent an m-dimensional vector field with m distinct scalar fields. In practice, this is usually not an intuitive representation.

In specific cases, other ways of using color to represent vector fields may be useful. Consider, for instance, a vector field f representing the velocity of a fluid. Consider an isosurface $\mathcal{S}$ of the vector magnitude scalar field which is colored, at every point $p \in \mathcal{S}$, by the angle between $f(p)$ and normal $n(p)$ of $\mathcal{S}$ at point p. This representation is useful to study how much the flow direction differs from the direction of the isosurface normals.

Let $\mathcal{S} \subseteq D$ be a surface represented as a discrete unstructured grid G and $f : D \to D$ a vector field. For every vertex p_i of a cell in the grid G, let $p'_i = p_i + kf(p_i)$ and consider a grid G' with the same structure as G defined by the points p'_i. If f represents a quantity such as a velocity and the surface is interpreted as a set of spatial coordinates, a grid defined in this way intuitively corresponds to a displacement of every point in the original surface to the position it would occupy a *small* time step into the future. The constant k controls how pronounced the difference in position becomes in the deformed surface, which is called a displacement plot. It is important to note the difference between a height plot and a displacement plot: a height plot is a representation of a scalar field through deformation of a pre-defined surface in the direction of its normals, while a displacement plot is a representations of a vector field through deformation of a pre-defined surface in the direction given by the vector field. The value of the constant k needs to be chosen carefully, to avoid self-intersecting surfaces. Scaling and clamping can also be applied. It is also important to choose the surface to be deformed appropriately, which depends on the vector field being studied.

Vector fields may also be represented by stream objects. If a vector field $f : \mathbb{R}^3 \to \mathbb{R}^3$ is interpreted as a velocity field for each $p \in \mathbb{R}^3$, then a stream object can be interpreted as the trajectory that a particle released at a point p would follow in the vector field.

A streamline is a polyline defined by the set of pairs (p_i, p_{i+1}) such that $p_{i+1} = p_i + \Delta t f(p_i)$. The point p_0 is called seed and is defined by the user. The parameter Δt is a time discretization constant. Alternatively, if the vector field f is also a function of time, i.e. $f : \mathbb{R}^3 \times T \to \mathbb{R}^3$, then p_{i+1} can also be defined as $p_{i+1} = p_i + \Delta t f(p_i, i\Delta t)$.

Streamlines may be colored by any scalar field, including the magnitude of the vector field that they represent. A streamline can be traced until it leaves the vector field domain or a pre-defined time is exceeded. It is also possible to stop it when its length gets too large or when the magnitude of the vector field drops bellow a certain value. It is important to notice that the error in positioning the streamline vertices may increase with every new vertex.

The choice of Δt is crucial to the success of streamlines. Using a constant value corresponds to sampling the time dimension uniformly, which leads to non-uniform sampling of the spatial dimensions. An alternative is to use only the direction of $f(p_i)$ to guide the trajectory of the streamline, which corresponds to non-uniform sampling of the temporal dimension.

Another important choice is the positioning of the streamline seeds, which can be uniform, random, or based on other heuristics, such as the proximity to already existing streamlines.

A streamtube is the geometrical object obtained by sweeping a circular cross section across a streamline. Glyphs can be added to mark the beginning and end of a streamtube. The thickness of the tube can be used to convey yet another scalar value. The considerations about glyph occlusion and cluttering also apply to streamtubes.

A stream ribbon is another interesting way of representing a vector field. Two streamlines are launched from seed points close to each other, and a surface similar to a ribbon is created by connecting the pair of points to their correspondents in subsequent timesteps. Effects such as twisting are easily perceived in this representation.

The idea in texture based vector visualization techniques is to create a texture signal that encodes the direction and magnitude of a vector field.

One of the simplest texture based vector visualization techniques is line integral convolution. Given a (randomly generated) grey noise texture image, the technique produces another image that represents a two dimensional vector field. Each pixel in the output image is the result of the convolution of a pixel p in the original image. This convolution is the weighted sum of the pixels in a streamline traced (forwards and backwards in time) from p, where the weight is given by the distance (along the streamline) of p to that point. Intuitively, this blurs the pixels in the direction of the streamline, while leaving high frequencies orthogonal to the streamlines.

An alternative to line integral convolution is a family of techniques called image based flow visualization. This family of techniques produces an animated sequence of noise textures that represent the vector field, and is completely based on transformations over the current image shown in the frame buffer. The idea can be summarized as follows. Given a vector field defined over a two dimensional grid, move every point of the grid by a small constant Δt in the direction of the vector field at that point. In the first iteration, the frame buffer should be initialized to a random noise image. After that, at each iteration, the frame buffer is read and used to texture the deformed grid with opacity $1 - \alpha$. This texture is added to another noise image with opacity α, which is a periodic function of time and the initial random noise image. Intuitively, the particles in the texture keep moving in the direction of the vector field, and new particles appear without interrupting this pattern due to the periodicity of the random noise.

Important parameters for image based flow visualization are the size of the spots in the noise image, the blending parameter α and the time step Δt.

7 Volume visualization

This section presents techniques for the visualization of volumetric data. Such data is represented by a function $f : D \to C$, where $D \subseteq \mathbb{R}^3$ and $C \subseteq \mathbb{R}$.

Using standard color mapping techniques, it is possible to create a naive two dimensional rendering of such data. However, this only allows the visualization of values assigned to elements in the boundary of the domain. Another option is to show slices of the data, which can be interactively explored by the user. This alternative fails in giving an overview of all the data in a single image. Another possibility is computing isosurfaces for a given isovalue, and rendering the resulting grid.

A third option is to visualize m semi-transparent slice planes in the same image. Computing and rendering slices that are orthogonal to one of the standard axis given a volumetric *image* is trivial. However, this is not useful if the viewing direction is also orthogonal to that standard axis. Alternatively, the slice planes may be orthogonal to the viewing direction, in which case that problem does not occur. This requires the recomputation of the slices every time the viewing direction changes, and also introduces some particular artifacts related to the number of slices that a ray coming from the camera intersects.

Volume visualization can be accomplished by ray tracing. Consider the scalar function $v : D \to \mathbb{R}$, where $D \subseteq \mathbb{R}^3$. The goal of ray tracing is to define an image $I : D' \to \mathbb{R}$, where $D' \subseteq \mathbb{R}^2$. For a given image pixel $p' \in D'$ and its corresponding point p in the user-defined image plane inside D, this is accomplished by tracing a ray r, that starts at p and is orthogonal to the image plane, that may intersect D. The first point in D to be intersected by such ray is denominated q_0 and the first point to leave D is q_1. We assume that D is a convex set. The ray is also associated to a function $q(t) = (1 - \frac{t}{T})q_0 + \frac{t}{T}q_1$, with $T = ||q_1 - q_0||$ and $t \in [0, T]$. Intuitively, the parameter t indicates the distance along the ray from the entry point q_0, and $q(t)$ is the corresponding element in D. We also define $s : [0, T] \to \mathbb{R}$ to be $s(t) = v(q(t))$.

Finally, we define $F : \mathcal{F} \to \mathbb{R}$, where $\mathcal{F}$ is the set of functions from $[0, T]$ to $\mathbb{R}$. The function F is called a ray function, and is responsible for associating a scalar value to a pixel given all scalar values along the respective ray.

It is common to combine ray functions with transfer functions, which map scalar values to colors (including opacity). A transfer function can be represented as $f : \mathbb{R} \to [0, 1]^4$ or, alternatively, by its four components (in the

RGBA colorspace) f_R, f_G, f_B and f_A (opacity). The image I resulting from this combination can be written as

$$I(p') = f(F(s)).$$

The process of designing transfer functions for visualization purposes is called classification and may be notoriously difficult in some applications of practical interest.

The maximum intensity ray function for a pixel p, given its ray r and function that gives the scalar values along the ray s, is defined as $F_{\max}(s) = \max_{t\in[0,T]} s(t)$. The value associated to a pixel is, therefore, the maximum scalar value along its ray. This is useful to emphasize high-valued points in the volume. The problem with this ray function is that it loses depth information.

The average intensity ray function is defined as $F_{\text{avg}}(s) = \frac{\int_{t=0}^{T} s(t)dt}{T}$. Intuitively, it gives the average of the scalar values along the ray.

The distance to value ray function is defined as $F_{\text{dtv}}(s) = \min_{t\in[0,T], s(t)\geq\sigma} t$. Intuitively, it gives the minimum value of t at which a certain value σ is exceeded when considering the scalar values along the ray.

The isosurface ray function is defined as

$$F_{\text{iso}}(s) = \begin{cases} \sigma & \text{if } \exists t \in [0,T] \text{ such that } s(t) = \sigma, \\ \sigma_0 & \text{otherwise.} \end{cases}$$

Therefore, a pixel is associated to scalar value σ if there exists, along its ray, a point with associated scalar value σ. Otherwise, it is associated to the scalar value σ_0. This ray tracing technique is usually combined with volumetric shading, otherwise the resulting image would be created from only two scalar values.

In *composite* ray tracing, the image I is defined as

$$I(p') = H(f \circ s),$$

where H is called a composite ray function such that $H : \mathcal{H} \to [0,1]^4$, where $\mathcal{H}$ is the set of functions from $[0,T]$ to $[0,1]^4$, f is a transfer function and s is the function that associates a scalar value to every $t \in [0,T]$ along the ray associated to point p. The idea in composite ray tracing is to first apply the transfer function to every scalar value along the ray and then combine them using a ray function. The composite ray tracing function usually computes, for every pixel p in the image, a color based on the attenuation of the colors along a the ray corresponding to p. The ray functions mentioned previously can also be implemented using composite ray tracing.

Composite ray tracing is a powerful method for creating visualizations that display several objects of interest from a given volume. The transfer function should be designed in such a way that particular scalar values of interest are assigned to high opacities and distinct colors, while uninteresting scalar values should be associated to low opacities.

Volumetric shading may be added to composite ray tracing. The idea is to apply the Phong lighting model to every point $q(t)$ in a ray, using an approximation to the normal at the isosurface defined by the isovalue $s(t)$ based on the gradient vector $\nabla s(t)$ and colors based on $f(s(t))$.

In practice, several choices must be made in order to implement ray tracing. For instance, the ray tracing functions need to work with rays discretized into a number of steps. The step size is a free parameter and, naturally, smaller steps give better results. Also, interpolation must be used to assign a color to every point in a ray from the original volumetric data set. If the original data set is a volumetric image, it may be necessary to convert from a reconstruction defined in terms of constant basis functions to a reconstruction based on trilinear basis functions.

License

References

[1] Telea, A. *Data Visualization, Principles and Practice*, 2014.