CPS 533 Scientific Visualization

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 7: Advanced Computer Graphics

- We introduced fundamental concepts of computer graphics in Chapter 3, where we discuss how to represent and render geometry using surface primitives such as points, lines, and polygons.

- In this chapter, we will discuss object transparency, texture mapping, and other volume graphics such as object-order and image-order techniques, illumination models, stereo viewing, antialiasing, and advanced camera techniques.
Opaque objects are the objects that reflect, scatter, or absorb light at their surface. Transparent objects are objects that transmit light.

In computer graphics, a property of alpha is introduced to indicate transparency and opacity. For example, a polygon that is 50 percent opaque will have an alpha value of 0.5 on a scale from zero to one. An alpha value of 1 represents an opaque object, and an alpha value of 0 represents a completely transparent object. The RGB specification of color is extended to RGBA where A represents the alpha component. The alpha value is stored with the RGB values in frame buffer on graphics cards.
In ray tracing, viewing rays are projected from the camera out into the world, where they intersect the first actor they come to. With an opaque actor, the lighting equations are applied and the resulting color is drawn to the screen. With a semitransparent actor we must solve the lighting equations for this actor. The resulting color is a composite of all the actors it has intersected.

\[
R = A_s R_s + (1 - A_s) R_b \\
G = A_s G_s + (1 - A_s) G_b \\
B = A_s B_s + (1 - A_s) B_b \\
A = A_s + (1 - A_s) A_b
\]

In this equation subscript \( s \) refers to the surface of the actor, subscript \( b \) refers to what is behind the actor. The term \( 1 - A_s \) is called the transmissivity, and represents the amount of light that is transmitted through the actor.
When ray-tracing a scene, we will intersect the surfaces from front to back. We trace the ray back to the last surface it intersects, and then composite the color using the lighting equations to all the surface in reverse order, from back to front. In object-order rendering, this process is supported by hardware, but the we are not guaranteed to render the polygons in any specific order. In the above picture, the polygons are arranged blue, green, and red from back to front, but the rendering sequence may be blue, followed by red, and then green. The resulting color is different.

\[
\begin{align*}
A &= A_s + (1-A_s)A_b = 0.5 + (1-0.5) \times 0 = 0.5 \\
B &= A_sB_s + (1-A_s)B_b = 0.5 \times 0.8 + (1-0.5) \times 0 = 0.4 \\
G &= A_sG_s + (1-A_s)G_b = 0.5 \times 0 + (1-0.5) \times 0 = 0 \\
R &= 0 \\
(0, 0, 0.4, 0.5)
\end{align*}
\]

\[
\begin{align*}
A &= A_s + (1-A_s)A_b = 0.5 + (1-0.5) \times 0.5 = 0.75 \\
B &= A_sB_s + (1-A_s)B_b = 0.5 \times 0.0 + (1-0.5) \times 0.4 = 0.2 \\
G &= A_sG_s + (1-A_s)G_b = 0.5 \times 0.8 + (1-0.5) \times 0 = 0.4 \\
R &= 0 \\
(0, 0.4, 0.2, 0.75)
\end{align*}
\]

\[
\begin{align*}
A &= A_s + (1-A_s)A_b = 0.5 + (1-0.5) \times 0.75 = 0.875 \\
B &= A_sB_s + (1-A_s)B_b = 0.5 \times 0.0 + (1-0.5) \times 0.2 = 0.1 \\
G &= A_sG_s + (1-A_s)G_b = 0.5 \times 0.8 + (1-0.5) \times 0 = 0.4 \\
R &= A_sR_s + (1-A_s)R_b = 0.5 \times 0.8 + (1-0.5) \times 0 = 0.4 \\
(0.4, 0.2, 0.1, 0.875)
\end{align*}
\]
When the blue polygon is rendered, the frame buffer and z-buffer are empty, so the RGBA quad \((0,0,0.8,0.5)\) is stored together with its z-buffer value. When the red polygon is rendered, a comparison of its z-value and the current z-buffer indicates that it is in front of the previous pixel entry. We use lighting equations to calculate resulting RGBA value, which is written to the buffer. Now the green polygon is rendered and the z comparison indicates that it is behind the current pixel’s value. We use the lighting equations to calculate the resulting RGBA value, where we use the frame buffer’s RGBA value for the surface and the polygon’s value for the behind. Remembering, once the red and blue polygons have been composited and written to the frame buffer, there is no way to insert the final green polygon into the middle where it belongs.
One solution to this problem is to sort the polygons from back to front and then render them in this order. This will bring additional computational overhead.

Another solution is to store more than one set of RGBA values in the frame buffer. This requires additional memory cost.

Numerical errors could be a problem in volume rendering. In volume rendering, it is desirable to have thousands of polygons with small alpha values. If the RGBA quad is stored in the frame buffer as four eight-bit values, then the round-off errors can accumulate over many polygons, resulting in large errors in the output images.
Texture mapping is a technique to add an image without requiring modeling detail. Two pieces of information are required for texture mapping, a texture map and texture coordinates. The texture map is the picture that is to be pasted, and the texture coordinates specify the location where the picture is pasted.

Texture mapping is a lookup table for color, intensity, and/or transparency that is applied to an object when it is rendered.

There are two ways of dealing with the original color of the object in texture mapping: (a) to ignore the original color, and apply the texture color as specified; (b) to modulate the original color by the texture map color or intensity to produce the final color.

For each pixel in the texture map, or a texel for texture element, there are one to four components, they are intensity map, RGB triplets, and alpha value.
In texture mapping, we need to determine how to map the texture onto the geometry. Each vertex has an associated texture coordinate in addition to its position, surface normal, color, and other point attributes. The texture coordinate system uses the parameters $(u,v)$ and $(u,v,t)$ for specifying 2D and 3D texture values.
We can also use procedural texture definitions to do texture mapping. In this approach, instead of using the \((u,v,t)\) texture coordinates to index into an image, we pass \((u,v,t)\) as arguments to the procedural texture that uses them to calculate a texel value.

Texture maps can be generated procedurally as a function of data, for example, to change the appearance of a surface based on local data value. We can threshold geometry by creating a special texture map and then setting texture coordinates based on local data value. The texture map consists of two entries, fully transparent \((\alpha=0)\) and fully opaque \((\alpha=1)\). The texture coordinate is then set to index into the transparent portion of the map if the scalar value is less than some threshold, or into the opaque portion otherwise.

Texture maps can be animated as a function of time. By choosing a texture map whose intensity varies monotonically from dark to light, and then moving the texture along an object, the object appears to crawl in the direction of the texture map motion. We can use this technique to add apparent motion to things like hedgehogs to show vector magnitude.
7.3 Volume rendering

- Some applications require us to visualize data that is inherently volumetric. In biomedical imaging, we may need to visualize data obtained from an MRI or CT scanner, a confocal microscope, or an ultrasound study.

- Volume was defined as a process that operates directly on the dataset to produce an image without generating an intermediate geometric representation, and now is defined as any method that operates on volumetric data to produce an image.

- In an image-order method, rays are cast for each pixel in the image plane through the volume to compute pixel values, while in an object-order method the volume is traversed, typically in a front to back or back to front order, with each voxel processed to determine its contribution to the image.
7.4 Image-order volume rendering

Image-order volume rendering is often referred to as ray casting or ray tracing. The basic idea is that we determine the value of each pixel in the image by sending a ray through the pixel into the scene according to the current camera parameters. Then we evaluate the data encountered along the ray using some specified function in order to compute the pixel. Ray tracing is a flexible technique that can be used to render any structured points dataset, and can produce a variety of images.

In standard parallel projection, rays are parallel to each other and perpendicular to the view plane. The data values along each ray are processed according to the ray function, which in this case determines the maximum value along the ray and converts it to a gray scale pixel value where the minimum scalar value in the volume maps to transparent black, and the maximum scalar value maps to opaque white.

The two main steps of ray tracing are determining the values encountered along the ray, and then processing these values according to a ray function.
This is the data value profile of a ray as it passes through 8 bit volumetric data where the data values can range between 0 and 255. The x-axis of the profile indicates distance from the view plane and the y-axis represents data value.

In the example, the first two ray functions, maximum value and average value, are basic operations on the scalar values themselves. The third ray function computes the distance along the ray at which a scalar value at or above 30 is first encountered, while the fourth uses an alpha compositing technique, treating the values along the Ray as samples of opacity accumulated per unit distance.

The maximum intensity projection, MIP, is probably the most common way to visualize volumetric data. A problem with method is that it is not possible to tell from a still Image where the maximum value occurred along the ray.
A volume is represented as a 3D structured points dataset where scalar values are defined at the points of the regular grid, while in ray tracing we often need to sample the volume at arbitrary locations. We do this by defining an interpolation function that can return a scalar value for any location between grid points. The simplest interpolation, which is called zero-order, constant, or nearest neighbor interpolation, returns the value of the closest grid point. This function defines a grid of identical rectangular boxes of uniform value centered on grid points. In the right image, we see an example of trilinear interpolation where the value at some location is defined by using linear interpolation based on distance along each of the three axes.
To traverse the data along a ray, we could sample the volume at uniform intervals or we could traverse a discrete representation of the ray through the volume, examining each voxel encountered. The selection of a method depends on factors such as the interpolation technique, the ray function, and the trade-off between image accuracy and speed.

The light ray is typically represented in parametric form as

\[(x, y, z) = (x_0, y_0, z_0) + (a, b, c)t\]

where \((x_0, y_0, z_0)\) is the origin of the ray (either the camera position for perspective transformations or a pixel on the view plane for parallel transformations), and \((a, b, c)\) is the normalized ray direction vector.
t = t1
v = undefined
While ( t < t2)
{
x = x0 + a*t;
y = y0 + b*t;
z = z0 + c*t;
v = EvaluateRayFunction(v, t);
t = t + delta_t;
}

This piece of code is to create uniform distance sampling. If \( t_1 \) and \( t_2 \) represent the distances where the ray enters and exits the volume respectively, and \( \Delta t \) indicates the step size. One difficulty with the uniform distance sampling is selecting the step size. If the step size is too large, then our sampling might miss features in the data, on the other hand, if we select a small step size, the amount of time for rendering the image will be significantly increased.
In some cases, it makes more sense to examine each voxel along the ray rather than taking samples. A 3D scan conversion technique can be used to transform the continuous ray into a discrete representation. The discrete ray is an ordered sequence of voxels $v_1, v_2, \ldots, v_n$, and can be classified as 6-connected, 18-connected, and 26-connected. Each voxel contains 6 faces, 12 edges, and 8 vertices. If each pair of voxel $v_i, v_{i+1}$ along the ray share a face then the ray is 6-connected, if they share a face or an edge the ray is 18-connected, if they share a face, an edge, or a vertex the ray is 26-connected. Scan converting and traversing a 26-connected ray requires less time than a 6-connected ray but is more likely to miss small features in the volume dataset.
If we are using a parallel viewing transformation and our ray function can be efficiently computed using a voxel traversal method, then we can apply a templated ray-casing technique with 26-connected rays to generate the image. Since all rays are identical in direction, we only need to scan convert once, using the template for every ray. When these rays are cast from pixels on the image plane, some pixels in the dataset will not contribute to the image. Instead, if we do the ray tracing from the voxels in the base plane of the volume that is most parallel to the image plane, the rays fit well such that every voxel in the dataset is visited exactly once. Because it is generated from the base plane, the image appears warped, a final resampling step is required to project the image back onto the image plane.
Object-order volume rendering methods process samples in the volume based on the organization of the voxels in the dataset and the current camera parameters. The voxels must be traversed in either a front to back or back to front order to obtain correct results. When implemented in hardware, a back to front ordering is preferred, when implemented in software, a front to back ordering is more common.

For an object-order, back to front ordering, voxel traversal starts at the voxel that is furthest from the view plane and then closer voxels until all voxels have been visited. This is done by a triple nested loop, from the outer to the inner loop, the planes in the volume are traversed, the rows in a plane are processed, and finally the voxels along a row are visited.

When a voxel is processed, its projected position on the view plane is determined and an operation is performed at that pixel location using the voxel and image information. This process may cause image artifacts due to the discrete selection of the projected image pixels. For example, when we move camera closer to the volume in a perspective projection, neighboring voxels will project to increasingly distant pixels on the view plane, resulting in distracting “holes” in the image.
This figure shows a simple object-order, back-to-front approach to projecting the voxels in a volume for an orthographic projection. The ordered labeling of the first sixteen voxels of the volume is projected. Processing voxels in this manner does not yield a strict ordering from the furthest to the closest voxel. However, it is sufficient for orthographic projections since it does ensure that the voxels that project to a single pixel are processed in the correct order.
Volume rendering by texture mapping

- Two-dimensional texture-mapped volume rendering makes use of 2D texture mapping hardware, 3D texture-mapped volume rendering makes use 3D texture mapping graphics hardware.

- In texture mapping, there are two basic steps, sampling step where data samples are extracted by interpolation, and blending step where the sample values are combined with the current image in the frame buffer.

- Texture-mapped volume renderers sample and blend a volume to produce an image by projecting a set of texture-mapped polygons of a volume.
In 2D texture-mapped volume rendering the dataset is decomposed into a set of orthographic slices along the axis of the volume most parallel to the viewing direction.

The slices are looped in a back to front order, where for each slice, a 2D texture is downloaded into texture memory. Each slice, which is a rectangular polygon, is projected to show the entire 2D texture. If neighboring slices are far apart compared to the image size, then it is necessary to use a software bilinear interpolation method to extract additional slices from the volume in order to achieve a desired image accuracy.

The texture download rate is the rate at which this image can be transferred from the main memory to texture mapping memory.
7.7 Volume classification

Classifying the objects of interest within a dataset is a critical step in producing a volume rendered image. This information is used to determine the contributions of an object to the image as well as the object’s material properties and appearance.

For example, in a CT scan, how can we distinguish bone from tissue? We can use a simple binary classification for the data sample. We use density, a scalar, and specify a density threshold to classify bone and tissue.
A transfer function maps the information at a voxel location into different values such as material property, color, or opacity. The transfer function can be a binary function, or gradual transformation function that describes graduate transition of different materials. Classifying a volume based on scalar value alone is often not capable of isolating an object of interest. A gradient magnitude is frequently added to the transfer function. A gradient is a vector, which has three components in three dimension. We can specify an object in the volume based on a combination of scalar value and the gradient magnitude. This allow us to define an opacity transfer function that can target voxels with scalar values in a range of densities and gradients within a range of gradient magnitudes. This is also useful for avoiding the selection of homogeneous regions in a volume and highlighting fast-changing regions. Due to this technique, we can visualize the sharp changes in the volume, such as the transition from air to skin and flesh to bone. For the homogeneous regions, such as internal muscle, it is often visualized as transparent.
Finite difference expression of gradient vector is:

\[ g_x = \frac{f(x + \Delta x, y, z) - f(x - \Delta x, y, z)}{2\Delta x} \]

\[ g_y = \frac{f(x, y + \Delta y, z) - f(x, y - \Delta y, z)}{2\Delta y} \]

\[ g_z = \frac{f(x, y, z + \Delta z) - f(x, y, z - \Delta z)}{2\Delta z} \]

where \( f(x,y,z) \) represents the scalar value at location \((x,y,z)\) in the dataset according to the interpolation function, and \( g_x, g_y, \) and \( g_z \) are the partial derivatives of this function along the \( x, y, \) and \( z \) axes respectively. The magnitude of the gradient at \((x, y, z)\) is the length of the resulting vector \((g_x, g_y, g_z)\). This vector can also be normalized to produce a unit normal vector. The choice of \( \Delta x, \Delta y, \) and \( \Delta z \) are critical. If these values are too small, then the computed gradient vector field may contain high frequencies, if these values are too large we will lose small features in the dataset.
7.8 Volumetric illumination

- In a maximum intensity projection, a dark region in the image indicates the lack of high opacity values in the corresponding region of the volume, while a dark feature in a shaded image may indicate either low opacity values or values with gradient directions that point away from the light source.

- The process of volume rendering is to create a 2D image from 3D data, and viewers of the image should be able to understand the 3D structure of the volume from the image.

- A static image showing a maximum intensity projection does not include occlusion or lighting effects, making it difficult to understand the structure. An image generated with a compositing technique includes occlusion, and the compositing ray function can be modified to include shading as well.
If the image that is produced as a result of volume rendering contains the distance from the view plane to the surface for every pixel, we can then post-process the image with a 2D gradient estimator to obtain surface normals. The gradient at a pixel \((x_p, y_p)\) can be estimated with a central difference as

\[
\frac{\partial Z}{\partial x} = \frac{Z(x_p + \Delta x, y_p) - Z(x_p - \Delta x, y_p)}{2\Delta x}
\]

\[
\frac{\partial Z}{\partial y} = \frac{Z(x_p, y_p + \Delta y) - Z(x_p, y_p - \Delta y)}{2\Delta y}
\]

\[\frac{\partial Z}{\partial z} = 1\]

The results are normalized to produce a unit normal vector. In the above equations, \(\Delta x\) and \(\Delta y\) are simply the pixel spacing in \(x\) and \(y\) directions. That is to say, we use the neighboring pixel values to estimate the gradient.
One problem of the 2D gradient estimation technique described above is that normals are computed from depth values that may represent disjoint regions in the volume. This may lead to a blurring of sharp features on the edges of objects. To reduce this effect, we can locate regions of continuous curvature in the depth image, then estimate the normal for a pixel using only other pixel values that fall within the same curvature region. We can achieve this by using small $\Delta x$ and small $\Delta y$. 

Disjoint volumetric objects

Corresponding depth image
For the discontinuous part, central difference may not be a good approximation. Instead, we use one-sided difference to calculate the gradient.

**Forward difference:**

\[
\frac{\partial Z}{\partial x} = \frac{Z(x_p + \Delta x, y_p) - Z(x_p, y_p)}{\Delta x}
\]

**Backward difference:**

\[
\frac{\partial Z}{\partial x} = \frac{Z(x_p, y_p) - Z(x_p - \Delta x, y_p)}{\Delta x}
\]
For both classification and illumination, we can either compute the scalar at any arbitrary location in the volume or compute the scalar at the grid points and the interpolate. If we do interpolation from the grid points, we can precompute the gradients for the entire dataset once, and store it for both classification and illumination. But the memory cost is very high. For a dataset with $256^3$ one-byte scalars, the initial storage is 16 Mbytes, the total storage will be 208 Mbytes, if we precompute the gradient and store them, assuming each component of the gradient is stored as a floating-point value, which occupies 4 bytes. Why?

To reduce the storage requirements, we could quantize the precomputed gradients by using some number of bits to represent the magnitude of the vector, and some other number of bits to encode the direction of the vector. Quantization works well for storing the magnitude of the gradient, but does not provide a good distribution of directions if we simply divide the bits among the three components of the vector.
A better approach is to use the uniform fractal subdivision of an octahedron into a sphere as the basis of the direction encoding. The vector directions encoded in this representation are all directions formed by creating a ray originated at the sphere’s center and passing through a vertex by creating a ray originating at the sphere’s center and passing through a vertex of the sphere. First we push all vertices back onto the original faces of the octahedron, then we flatten this sphere onto the $z=0$ plane. Finally, we rotate the resulting grid by $45^\circ$. 
The vertices in the grid are labeled with indices starting at 0 at the top left vertex, colored with red, and continue across the rows then down the columns to index 40 at the lower right vertex, colored with blue. These indices represent only half of the encoded normals because when we flattened the octahedron, we placed two vertices on top of each other on all but the edge locations. Thus, we can use indices 41 through 81 to represent vectors with a negative z component. Vertices on the edges represent vectors with one z component, and we could represent them with a single index, however, using two keeps the indexing scheme more consistent and easier to implement.

There are 66 unique vector directions, for a sphere at recursion level 2 (the board points are counted one, and inner points are counted twice, $16+25\times 2 = 66$), and we represent the with 82 values. For a recursion depth of 6, there are 16,386 unique directions and $16,386+256 = 16,642$ vertices.
7.9 Region of interests

One difficulty in visualizing volumetric data with the volume rendering methods presented so far is that in order to study some feature in the center of the volume we must look through other features in the dataset. We can solve the problem of visualizing internal features by defining a region of interest within our volume, and rendering only this portion of the dataset.

Techniques for defining region of interest include using the near and far clipping planes of the camera to exclude portions of the volume, using six orthographic clipping planes to define a rectangular subvolume, using a set of arbitrarily oriented half-space clipping planes, defining the region of interest as the portion of the volume contained within some set of closed geometric objects, and creating an auxiliary structured points dataset with binary scalar values that define a mask indicating which values in the volume should be considered during rendering.

For an image-order ray-tracing approach, the ray is clipped against all geometric region definitions, and the ray function is then evaluated only along segments of the ray that are within the region of interest. The mask values are consulted at each sample to determine if its contribution should be included or excluded.

For object-order methods, we must determine for each sample whether or not if is within the region of interest before incorporating its contribution into the image. If the underlying graphics hardware is being utilized for the object-order volume rendering as is the case with a texture mapping approach, hardware clipping planes may be available to help support regions of interest.
7.10 Intermixing volumes and geometry

- It can improve the viewer’s understanding of the volumetric data to visualize volumetric data using both geometric and volumetric methods within the same image.

- When using graphics hardware to perform volume rendering, as is the case with a texture mapping approach, intermixing opaque geometry in the scene is trivial. All opaque geometry is rendered first, then the semitransparent texture-mapped polygons are blended in a back to front order into the image. If we wish to include semitransparent geometry in the scene, then this geometry and the texture-mapped polygons must be sorted before rendering.

- If a software volume rendering approach is used, opaque geometry can be incorporated into the image by rendering the geometry, capturing the results stored in the hardware depth buffer, and then using these results during the volume rendering phase. For ray-tracing, we would simply convert the depth value for a pixel into a distance along the view ray and use this to bound the segment of the ray that we consider during volume rendering. The final color computed for a pixel during volume rendering is then blended with the color produced by geometric rendering.

- In an object-order method, we must consider the depth of every sample and compare this to the value stored in the depth buffer at each pixel within the image extent of this sample.
7.11 Efficient volume rendering

- Rendering a volumetric dataset is a computationally intensive task. If $n$ is the size of the volume on all three dimensions and we visit every voxel once during a projection, the complexity of volume rendering is $O(n^3)$. Even a highly optimized software algorithm will have great difficulty projecting a moderately sized volume of $512 \times 512 \times 128$ or approximately 32 million voxels at interactive rates. If every voxel in the volume contributes in some way to the final image and we are unwilling to compromise image quality, it difficult to improve efficiency. However, many volumetric datasets contain large regions of empty or uninteresting data that are assigned opacity values of 0 during classification.

- Space leaping refers to a general class of efficiency improvement techniques that attempts to avoid processing regions of a volume that will not contribute to the final image. One technique often used is to build an octree data structure which hierarchically contains all of the important regions in the volume. The root node of the octree contains the entire volume and has eight child nodes, each of which represents 1/8 of the volume. These eight subregions are created by dividing the volume in half along the x, y, and z axes. This subdivision continues recursively until a node in the octree represents a homogeneous region of the volume. With an object-order rendering technique, only the nonempty leaf nodes of the octree would be traversed during rendering hence avoiding all empty regions while efficiently processing all contributing homogeneous regions.
7.13 Volume rendering future

- Volume rendering has evolved from a research topic to home computers.
- Challenges: the number of volumes is growing, time-dependent datasets at discrete time intervals needs to be rendered.
- To extend the volume rendering techniques from structured points datasets to rectilinear grid, structured grid, and even irregular data.
Example of texture mapping
# include “vtk.h”
main()
{
    vtkRenderer *ren1 = vtkRenderer::New();
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin->AddRenderer(ren1);
    vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(renWin);
    vtkStructuredPointsReader *reader = vtkStructuredPointsReader::New();
    reader->SetFileName("../../../vtkdata/ironProt.vtk");
    reader->Update();
    vtkPiecewiseFunction *oTFun = vtkPiecewiseFunction::New();
    oTFun->AddSegment(80, 0.0, 255, 1.0);
    vtkPiecewiseFunction *gTFun = vtkPiecewiseFunction::New();
    gTFun->AddSegment(0, 1.0, 255, 1.0);
    vtkVolumeProperty *volProperty = vtkVolumeProperty::New();
    volProperty->SetColor(gTFun);
    volProperty->SetOpacity(oTFun);
    volProperty->SetInterpolationTypeToLinear();
    volProperty->ShadeOn();
    vtkVolumeRayCastCompositeFunction *compositeFunction = vtkVolumeRayCastCompositeFunction::New();
    vtkVolumeRayCastMapper *volMapper = vtkVolumeRayCastMapper::New();
    volMapper->SetScalarInput(reader->GetOutput());
    volMapper->SetVolumeRayCastFunction(compositeFunction);
    vtkVolume *vol = vtkVolume::New();
    vol->SetVolumeMapper(volMapper);
    vol->SetVolumeProperty(volProperty);
    ren1->AddVolume(vol);
    ren1->GetActiveCamera()->Azimuth(20);
    ren1->GetActiveCamera()->Dolly(1.65);

    iren->SetDesiredUpdateRate(3.0);
    iren->Start();
}
The following code segment is used to define a color transfer function ranging from red through blue to green. To produce a maximum intensity projection, we would simply change the type of the ray function to a vtkVolumeRayCastMIPFunction. We could also produce a surface image using a vtkVolumeRayCastIsosurfaceFunction where the IsoValue instance variable would be set to define the surface.

```cpp
tvtkColorTransferFunction *cTFun = vtkColorTransferFunction::New();
cTFun->AddRedSegment(0, 0.0, 64, 1.0);
cTFun->AddRedSegment(64, 1.0, 128, 0.0);
cTFun->AddBlueSegment(64, 0.0, 128, 1.0);
cTFun->AddBlueSegment(128, 1.0, 192, 0.0);
cTFun->AddGreenSegment(128, 0.0, 192, 1.0);
cTFun->AddGreenSegment(192, 1.0, 255, 0.0);
```
# include "vtk.h"
main()
{
    //create the rendering objects;
    vtkRenderer *ren1 = vtkRenderer::New();
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin->AddRenderer(ren1);
    vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(renWin);
    //create the pipeline, ball and spikes;
    vtkSphereSource *sphere = vtkSphereSource::New();
    sphere->SetThetaResolution(7); sphere->SetPhiResolution(7);
    vtkPolyDataMapper *sphereMapper = vtkPolyDataMapper::New();
    sphereMapper->SetInput(sphere->GetOutput());
    vtkActor *sphereActor = vtkActor::New();
    sphereActor->SetMapper(sphereMapper);
    vtkConeSource *cone = vtkConeSource::New();
    cone->SetResolution(5);
    vtkGlyph3D *glyph = vtkGlyph3D::New();
    glyph->SetInput(sphere->GetOutput());
    glyph->SetSource(cone->GetOutput());
    glyph->SetVectorModeToUseNormal();
    glyph->SetScaleModeToScaleByVector();
    glyph->SetScaleFactor(0.25);
    vtkPolyDataMapper *spikeMapper = vtkPolyDataMapper::New();
    spikeMapper->SetInput(glyph->GetOutput());
    vtkActor *spikeActor = vtkActor::New();
    spikeActor->SetMapper(spikeMapper);
    ren1->AddActor(sphereActor);
    ren1->AddActor(spikeActor);
    ren1->SetBackground(0.2, 0.3, 0.4);
    renWin->SetSize(400, 400);
    //do the first render and then zoom in a little;
    renWin->Render();
    renWin->GetActiveCamera() -> Zoom(1.4);
    renWin->StereoRenderOn();
    renWin->SetStereoTypeToRedBlue();
    renWin->Render();
}
//changes and additions to the preceding example's source
vtkActor *spikeActor2 = vtkActor::New();
  spikeActor2->SetMapper(spikeMapper);
  spikeActor2->SetPosition(0,-0.7,0);
  sphereActor2->SetPosition(0,-0.7,0);
ren1->AddActor(sphereActor2);
ren1->AddActor(spikeActor2);

//zoom in a little
ren1->GetActiveCamera()->Zoom(1.5);

renWin->SetSubFrames(21);

for(i=0; i<=1.0; i=i+0.05)
{
  spikeActor2->RotateY(2);
  sphereActor2->RotateY(2);
  renWin->Render();
}

iren->Start();
//changes and additions to the preceding example's source

//set the actors position and scale

spikeActor->SetPosition(0, 0.7, 0);
sphereActor->SetPosition(0, 0.7, 0);
spikeActor2->SetPosition(0, -0.7, -10);
sphereActor2->SetPosition(0, -0.7, -10);
spikeActor2->SetScale(2, 2, 2);
sphereActor2->SetScale(2, 2, 2);

//zoom in a little
ren1->GetActiveCamera()->SetFocalPoint(0, 0, 0);
ren1->GetActiveCamera()->Zoom(4);
ren1->GetActiveCamera()->SetFocalDisk(0.05);

renWin->SetFDFrames(11);  
Example of a scene rendered with focal depth
renWin->Render();

iren->Start();
3D widgets are a subclass of vtkInteractorObserver meaning that they are associated with a vtkRenderWindow and observe events in the render window(). The startInteractionEvent turns the visibility of the streamlines on; the InteractionEvent causes the streamlines to regenerate themselves.