Editor: Mike Potel
www.wildcrest.com

Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations

Jennis Meyer-Spradow, Timo Ropinski, Jörg Mensmann, and Klaus Hinrichs
University of Münster

The amount of volume data requiring analysis is rising significantly in many domains. For example, in medicine, computed tomography and magnetic resonance imaging have changed from seldom-applied special technologies to standard diagnostic tools. Visualization of the acquired volume data is challenging. Scanners used in medical research can create more than 1,000 slices per scan pass, which aren't manageable with standard visualization techniques. Routine medical diagnostics also need new visualization techniques for fast, reliable analysis of the huge amounts of data. Important information must be depicted concisely. To determine what information is important depends on what the medical question is and which scanning technologies are used—effective visualizations are usually specialized.

To support interactive development of such specialized visualizations, our *Voreen* volume-rendering engine provides an rapid-prototyping environment for realizing GPU-based volume ray-casting techniques.[1] Voreen is an open source C++ framework consisting of *processors*—autonomous functional building blocks that each perform a specific task. Voreen's users can flexibly combine these processors into data-flow networks. The component-based architecture ensures that users can exploit the realized techniques in interactive applications without any performance penalty. Voreen provides processors for different volume-rendering and image-processing techniques. Users can extend it easily by integrating more processors—for example, to support a new volume-processing algorithm. Voreen supports several file formats for volume data—for example, DICOM, TIFF, or RAW. Users can easily modify processor properties through automatically generated GUI components and can use the designed visualizations through either a generic application provided by Voreen or customized applications. For more on rapid-prototyping environments, see the sidebar.

Managing Data and Execution

On a logical level, data flows from one Voreen processor to another via ports—*inports* for input and *outports* for output. The type of port determines which inport/outport pairs are compatible and can thus be connected. Subtypes give further hints about the transferred data content. Although specific types of connected ports must match, this doesn't apply to the subtypes.

Voreen's design is object-oriented (OO), but this doesn't apply to the architecture of OpenGL—the underlying graphics system. In a purely OO environment, objects would manage themselves by transferring control and data directly. However, because OpenGL isn't OO, developers must carefully trade off between achieving high graphics performance and following OO design principles. For efficiency, OpenGL's state is determined by global variables, and data are accessible from anywhere in an OpenGL program. Encapsulating such data into objects would significantly decrease performance. So, Voreen uses a central instance for managing graphics data and exchanges these data via references. The central controller also schedules processor execution.

Coprocessors Supporting Processors

In some situations, data transfer via ports isn't suitable. For example, if one consumer processor would like to use data that a producer processor provides, the data must be transferred from producer to consumer. This transfer requires a generic data representation. However, such a representation isn't always feasible, or creating it might be time-consuming. For specific restricted data usage, it's often sufficient if the consumer can access the data indirectly by calling a method of the producer. So, we extend the data-flow principle: not only can data be transferred between processors via ports, processors also can access data by calling the methods of special processors called *coprocessors*. A coprocessor class can be used to share functionality between different processor classes.

Processor Classes

Here, we describe some important processor classes.

 Published by the IEEE Computer Society

Related Work in Rapid-Prototyping Environments

Scientists frequently perform volume data analysis and visualization, and many libraries, applications, and rapid-prototyping environments (RPEs) have been developed for this task. Libraries offer much flexibility but require much experience and effort to be beneficial. Applications are mostly easy to handle but aren't extensible and are thus limited to the built-in tasks the developer has anticipated. RPEs combine the best of both approaches. They usually provide a set of modules, which users can combine to achieve their goals.

Many RPEs are available either commercially or for free—for example, Amira (www.amira.com), MeVisLab (www.mevislab.de), VisTrails (www.vistrails.org), and XIP (Extensible Imaging Platform; https://collab01a.scr.siemens.com/xipwiki). Ingmar Bitter and his colleagues have compared some of these packages.[1] Unlike Voreen (see the main article), these RPEs focus on the entire data analysis process; visualization is only the final step and sometimes a minor aspect. For visualization, they use mostly GPU-based slicing approaches. However, slicing has inherent disadvantages resulting mainly from the inflexibility of the volume's slice-by-slice traversal.

Ray casting solves these issues because it traverses the volume data set separately for each ray. So, it offers greater flexibility. The first GPU-based solutions for ray casting were quite slow. But with the huge improvement in GPU technology, ray casting can be extended with several enhancements and still reach interactive frame rates. MeVisLab contains a GPU-based ray-casting module that supports renderer reconfiguration on a per-object basis.

The widely used VTK (Visualization Toolkit; www.vtk.org) library supports GPU-accelerated volume rendering with slicing only. The same is true for other libraries such as MITK (Medical Imaging Interaction Toolkit; www.mitk.org) or Mercury's commercial Open Inventor implementation (www.vsg3d.com). Jesus J. Caban and his colleagues have compared several open source libraries and applications.[2]

References

1. I. Bitter et al., "Comparison of Four Freely Available Frameworks for Image Processing and Visualization That Use ITK," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 3, 2007, pp. 483–493.
2. J.J. Caban, A. Joshi, and P. Nagy, "Rapid Development of Medical Imaging Tools with Open-Source Libraries," *J. Digital Imaging*, vol. 20, no. 1, 2007, pp. 83–93.

Because Voreen is extensible, users can easily add new processors.

A `VolumeSetSource` processor lets other processors access a data set. `VolumeSelector` selects a specific volume in the data set and specifies as a parameter the desired modality or index (in the case of a series of volumes).

An `EntryExitPoints` processor creates images containing color-coded entry and exit points using a proxy geometry. The color of each proxy geometry vertex encodes the corresponding vertex position. So, the GPU automatically calculates the color-coded entry and exit points when rendering the proxy geometry.[1] For cube-shaped volumes, the proxy geometry is an RGB color cube.

Because the demands on a proxy geometry can differ, it isn't created in `EntryExitPoints` itself, but the processor calls a coprocessor to render a certain proxy geometry. A simple `ProxyGeometry` coprocessor stores and renders a volume's bounding box. By modifying the proxy geometry, users can change the part of the volume data to be visualized (see Figure 1). More complex effects can be achieved by applying a `ProxyGeometry` that uses a mesh representation and relaxes the relation between vertex positions and colors. So, keeping the colors constant and changing the vertex positions can produce a rendering of a deformed volume.

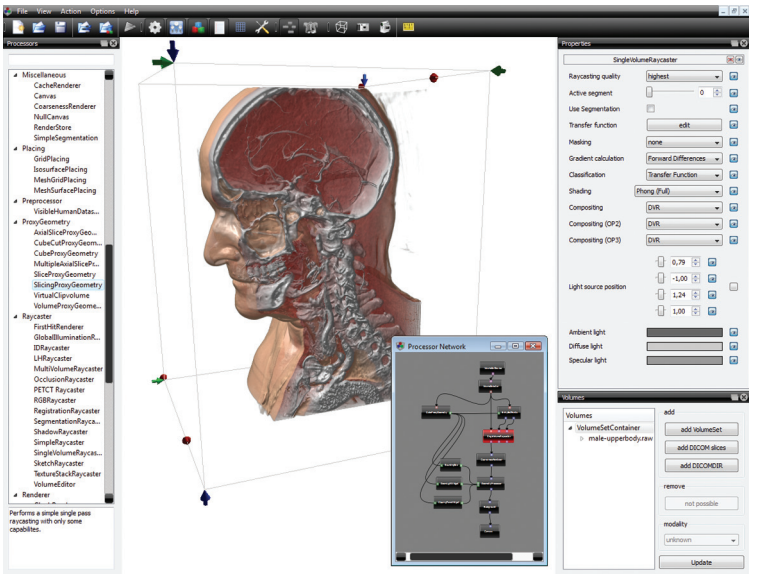


Figure 1. Rendering of a human head, with a clipped proxy geometry. You can download the visualization environment shown in this figure at www.voreen.org

`ProxyGeometry` coprocessors illustrate the advantage of interaction between a processor and a coprocessor via a method call. If we used ports instead, we'd need the specific data representation to be transferred. Not forcing `ProxyGeometry` to provide information in a certain format relieves

```
1  Grayscale::Grayscale()
2  {
3    createInport("image.input");
4    createOutput("image.outport");
5    program_=ShdrMgr.load("grayscale.frag");
6    program_->build();
7  }
```

(a)

```
1  Identifier Grayscale::getClassName()
2  {
3    return ("PostProcessor.Grayscale";
4  }
5
6  Processor* Grayscale::create()
7  {
8    return new Grayscale
9  }
```

(b)

```
1  void Grayscale::process(PortMapping* pm)
2  {
3    int source = pm->getTarget("image.input");
4    int dest = pm->getTarget("image.output");
5
6    glBindTexture(getGLTexTarget(source),
7                  getGLTexID(source));
8    setActiveTarget(dest);
9    program_->activate();
10   program_->setUniform("tex_", 0);
11   program_->setUniform("texSize_", size_);
12   renderQuad();
13   program_->deactivate();
14 }
```

(c)

```
1  uniform sampler2D tex_;
2  uniform vec2 texSize_;
3
4  void main()
5  {
6    vec4 rgba = texture2D(tex_,
7                  gl_FragCoord.xy/texSize_);
8    float g = 0.3*rgba.r + 0.59*rgba.g +
9              0.11*rgba.b;
10   gl_FragColor = vec4(g, g, g, rgba.a);
11 }
```

(d)

Figure 2. New processors can be integrated easily into Voreen. This example shows the realization of a simple grayscale image processor. Creating a processor that converts an RGB image to a grayscale image: (a) setting up the inports and outports, (b) adding the unique processor name and a `create()` method, (c) implementing the `process()` method, and (d) the shader program.

the implementation from any restrictions concerning the internal representation, provided that it can render the stored shape.

A `RayCaster` processor receives entry and exit points as well as a volume data set. It then performs the ray casting in a fragment program on the GPU using the OpenGL Shading Language (GLSL). It supports different rendering modes—for example, isosurface rendering, maximum-intensity projection, and direct-volume rendering—by recompiling the shader program at runtime. However, small modifications such as changing parameters—for example, thresholds—don't require recompilation because uniform variables can pass new values. Using OpenGL's multiple-render-targets extension, the same rendering pass can produce multiple output images.

Once `RayCaster` renders the volumes into an image, further processors can be applied. Whenever possible, the images contain not only the RGBA channels but also depth values. Most image processors are regular filters as known from image-processing applications. The filters often benefit if depth values are available—for example, for edge detection. Multiple images can be combined in several ways.

`GeometryProcessors` fuse images with renderings of polygonal objects. To save rendering passes, one processor processes multiple objects together, using different coprocessors for the actual renderings.

`Caching` and `Coarseness` processors increase a network's rendering speed. `Caching` stores the output image obtained from its predecessor until that image becomes invalid. This can improve interactivity for complex volume visualizations, for example, to edit annotations without rerendering everything on every frame. To guarantee immediate reaction to user input, `Coarseness` reduces its predecessors' image resolution while the user interacts with the scene. Usually, this processor is only used for time-consuming subnetworks, so that inexpensive rendering processors still display in full resolution.

Creating New Processors

Creating new Voreen processors is easy. To demonstrate the general principle, we create a simple processor that converts an RGB image into a grayscale image. The new processor class constructor sets up its inports and outports before loading and building the shader program (see Figure 2a).

To instantiate the new processor objects at runtime, `ProcessorFactory` requires a unique processor name and a `create()` method (see Figure 2b).

Figure 2c shows the `process()` method implementation. Lines 3 and 4 indicate the input and output images' storage locations. Lines 6 and 7 bind the input image as a texture to make it accessible in the shader program. The code then activates the output image as the destination for any OpenGL output. Lines 9–11 initialize the shader program, which line 12 executes indirectly by drawing a screen-aligned quad.

Figure 2d shows the shader program. Lines 6 and 7 read the current fragment's color; lines 8 and 9 convert that color into a grayscale value, which line 10 writes to the output image.

A Basic Ray-Casting Network

Figure 3 shows a network that performs volume rendering by using a proxy geometry to calculate the ray entry and exit points and then feeding them into the ray caster. The `VolumeSetSource` processor specifies the data set. Then, `VolumeSelector` selects from the data set the volume to be rendered and delivers it to `RayCaster` and the `ProxyGeometry` coprocessor, which adapts the proxy geometry's size to the data-set size. `EntryExitPoints` obtains viewport information from the camera and creates the textures for entry and exit points using `ProxyGeometry`. `RayCaster` uses these textures to produce a volume rendering. `GeometryProcessor` adds some polygonal objects by delegating their rendering to the `ClippingPlanes`, `BoundingBox`, and `LightSource` coprocessors, which retrieve the required geometric information by calling a method of `ProxyGeometry`. The light source and the colored bounding-box arrows are interactive elements. So, the user can move the light source or shift the axis-aligned clipping planes. Finally, `Background` creates a background for the volume rendering.

Application Examples

Here, we describe three visualization techniques from different domains that we realized with Voreen.

Automatic Label Placement

Medical visualizations widely use illustrations—for example, to communicate anatomical structures in medical textbooks. With the advent of high-resolution medical scanners, creating medical illustrations using acquired data has become more common. In such illustrations, textual annotations are important to add descriptive labels to the objects of interest.

If possible, a label should be fitted to the object's surface to allow immediate identification. We've proposed an algorithm that automatically places

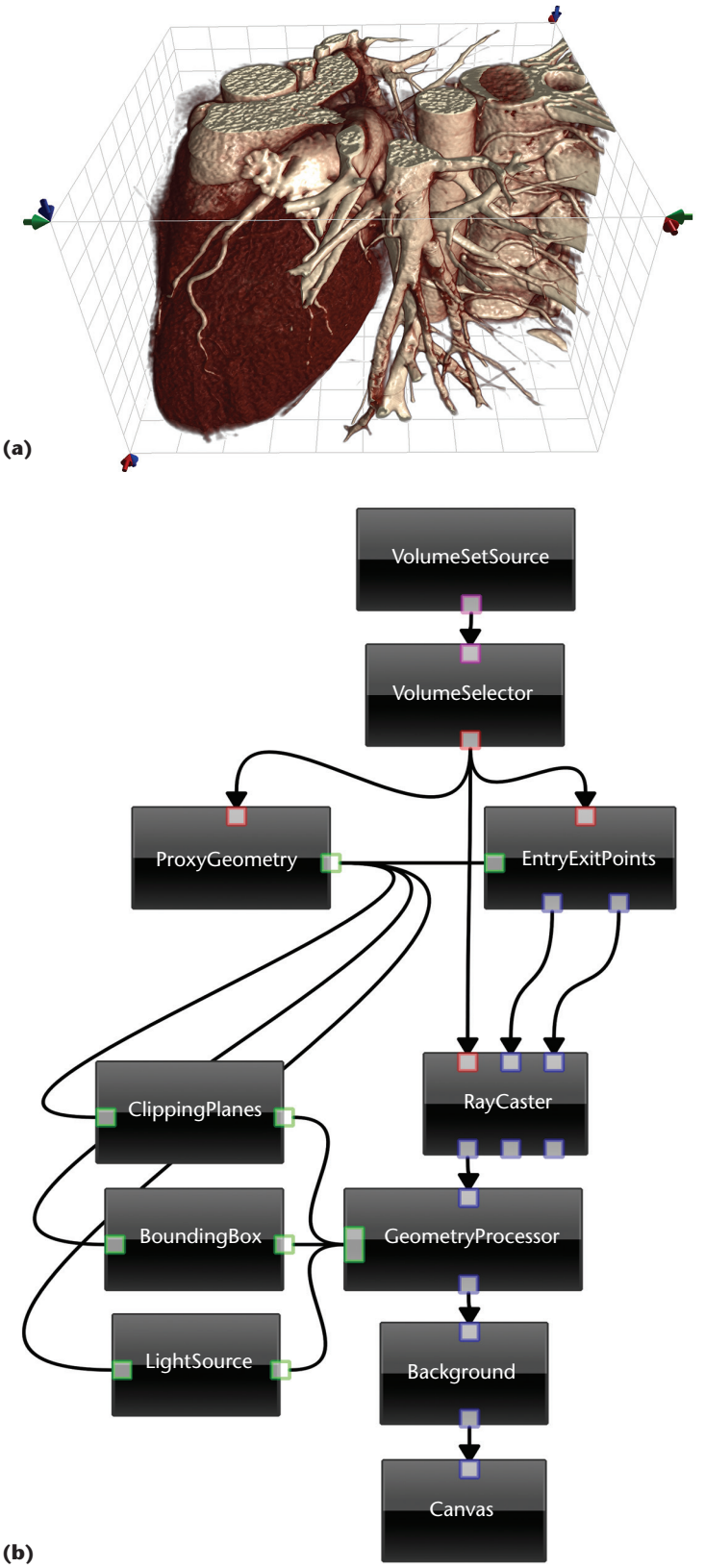
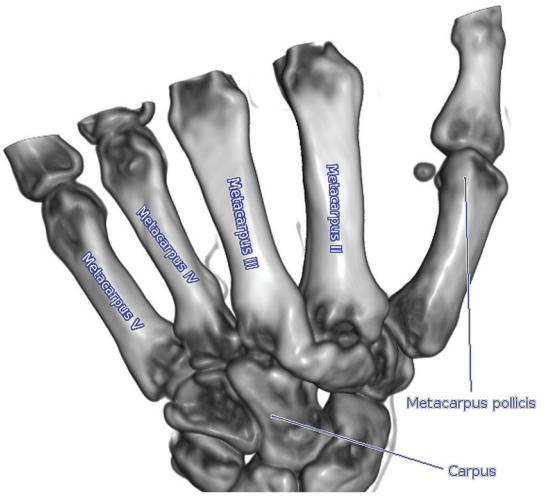
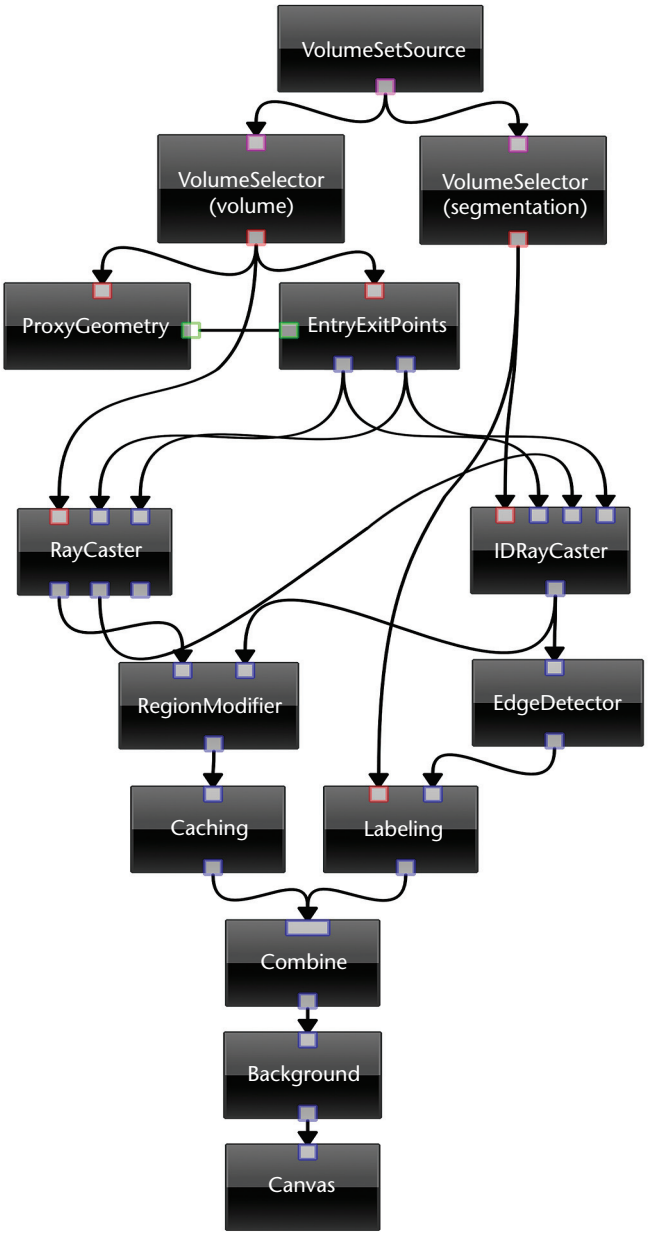


Figure 3. Networks can be generated by combining many existing processors. (a) To visualize a human heart, (b) a basic ray-casting network uses processors for generating a bounding box, as well as a clipping plane and a light-source widget (the colors of the ports indicate their type; red ports transmit volume data, and blue ports transmit images, whereas green ports indicate coprocessor connections).



(a)



(b)

Figure 4. Using Voreen for automatic label placement: (a) a labeled image of the human hand and (b) the corresponding data-flow network.

3D labels on the surfaces of objects in a segmented volume data set.[2]

The algorithm extends a 2D shape-fitting approach to ensure that a label's path matches the shape of the corresponding object's projection in image space. By analyzing the depth structure of the image, we extend the generated 2D path to a 3D path on the object's surface. The algorithm uses this 3D path to generate Bézier patches fitting the object's surface, on which the label is then rendered (see Figure 4a).

Figure 4b shows the network for the image in Figure 4a. For 2D shape fitting, the algorithm uses `ProxyGeometry`, `EntryExitPoints`, and `IDRayCaster` to render a segmented volume from the current view (from the right `VolumeSelector` in Figure 4b) into an ID map. `IDRayCaster` calculates the first-hit points in the volume and stores for each one the depth information and the color-coded unique ID assigned to the corresponding segment in the ID map. An object having front-facing parts partially covered by other parts of the object results in a segment having discontinuities in its depth values. To ensure that this object's label fits smoothly to its surface and doesn't cross such discontinuities, `EdgeDetector` performs depth-based contour detection to identify these discontinuities. Then, that processor splits the segment by changing the color-coded ID values corresponding to the discontinuous positions.

The resulting image goes to `Labeling`, which applies a distance transform to produce a distance map. For each pixel, this map contains the color-coded, closest distance to the segment border the pixel belongs to. `Labeling` then calculates the medial axes contained implicitly in the distance map and converts them to 2D curves. If `Labeling` is configured for 2D labeling, it renders the labels immediately along these 2D curves. For 3D labeling, `Labeling` can easily obtain the necessary 3D segments' structure if the ray casting's color-coded, first-hit positions are stored in a first-hit positions map. `IDRayCaster` can perform this action in parallel. `Labeling` identifies the 2D points of each medial axis with the corresponding points in the first-hit positions map, creates a 3D path for each medial axis, and approximates it with a Bézier spline. The surface in the proximity of each spline curve is then approximated with Bézier patches, onto which the label is rendered.

For visualization of the anatomical objects, our algorithm performs a regular ray casting using the actual volume (through `ProxyGeometry`, `EntryExitPoints`, and `RayCaster`). Optionally, `RegionModifier` can highlight a certain seg-

ment. `Caching` caches the result to prevent unnecessary renderings, when users drag labels with the mouse. Finally, `Combine` blends the images and `Background` adds a background.

Multivariate Visualizations Using Glyphs

Scientists widely use glyphs to visualize various data modalities simultaneously. For example, in a diagnosis of coronary artery disease based on SPECT (single photon emission computed tomography), physicians must consider various parameters such as blood supply under stress and rest or cardiac-wall parameters.

Figure 5 shows a glyph-based visualization that supports physicians performing diagnoses, and its corresponding network.[3] `GlyphPlacing` uniformly distributes glyphs over the cardiac wall's surface (provided by the right `VolumeSelector`). The `GlyphGenerator` coprocessor determines each glyph's shape and color to depict cardiac parameters (taken from the left and middle `VolumeSelector` processors) at the glyph's position. Using a coprocessor for glyph generation lets us provide different implementations for different glyph types. `Combine` blends the glyph image with a rendering of the cardiac wall's surface (created by `MeshRenderer`), and `Background` adds a background.

In this network, geometric objects (that is, vertices and edges), rather than volumetric data, are transferred between processors. Processors supporting glyphs in combination with volume rendering are also available.

Visualizing Motion in Still Images

Scientists often visualize time-varying volume data sets by displaying the 3D volumes sequentially in the order in which the volumes were acquired. Although rendering the data sets at high frame rates lets viewers construct a mental image of the temporal changes, in many situations, viewing a static image is more convenient.

We've introduced three techniques for visualizing dynamics in a single still image.[4] Figures 6 and 7 illustrate these techniques.

The first technique overlays a 3D volume rendering with silhouettes of the preceding and successive 3D volumes (see Figure 6a). Because we must visualize subsequent volumes of the same data set, we use a common `ProxyGeometry` and `EntryExitPoints` (see Figure 6c). The middle `RayCaster` generates the image for the current time step (taken from the middle `VolumeSelector`). To obtain the silhouettes, the left and right `RayCasters` perform the ray casting, and the two `EdgeDetectors` apply edge detection. (For brevity, we simplified the

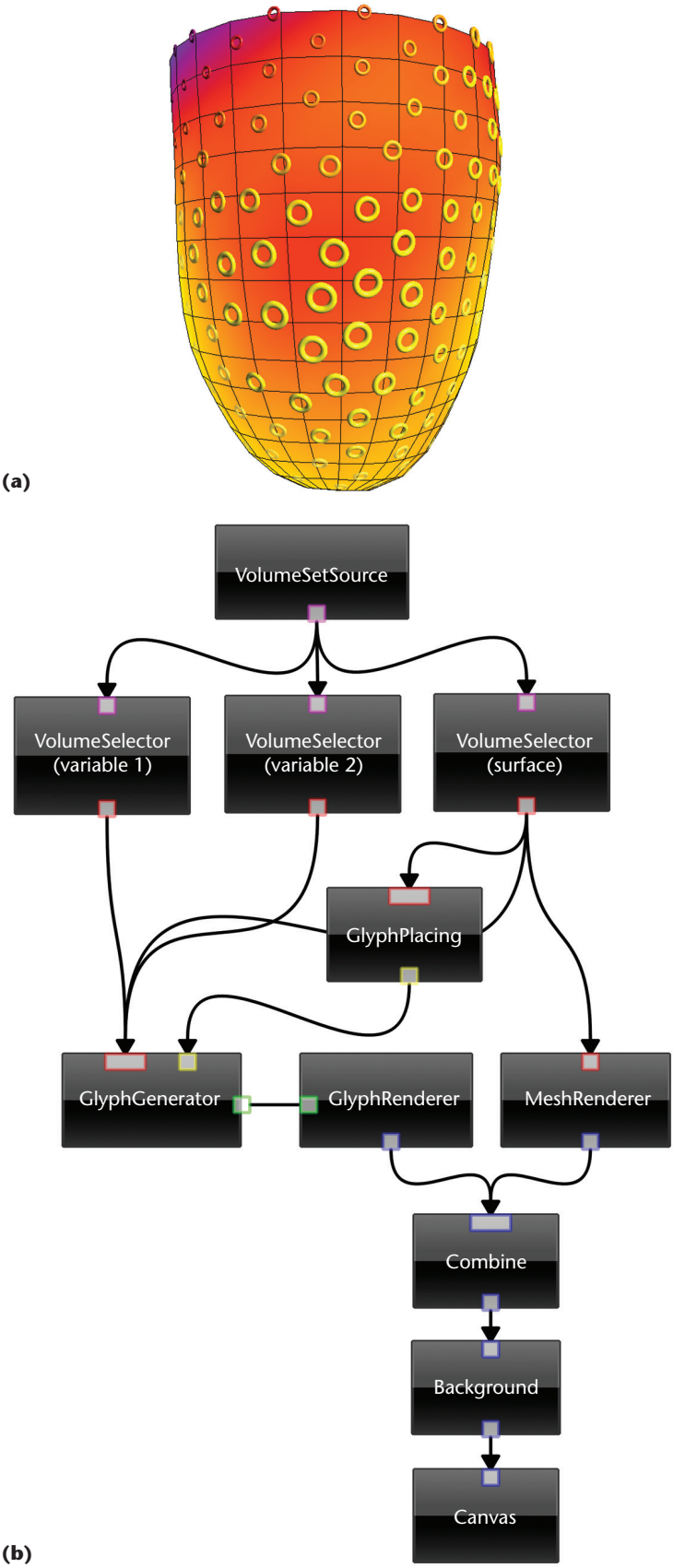


Figure 5. Voreen's functionality also lets you easily prototype visualizations containing mesh data. In this example case, (a) glyphs are used to visualize multiple data values simultaneously to help physicians diagnose coronary artery disease, and (b) shows the corresponding network.

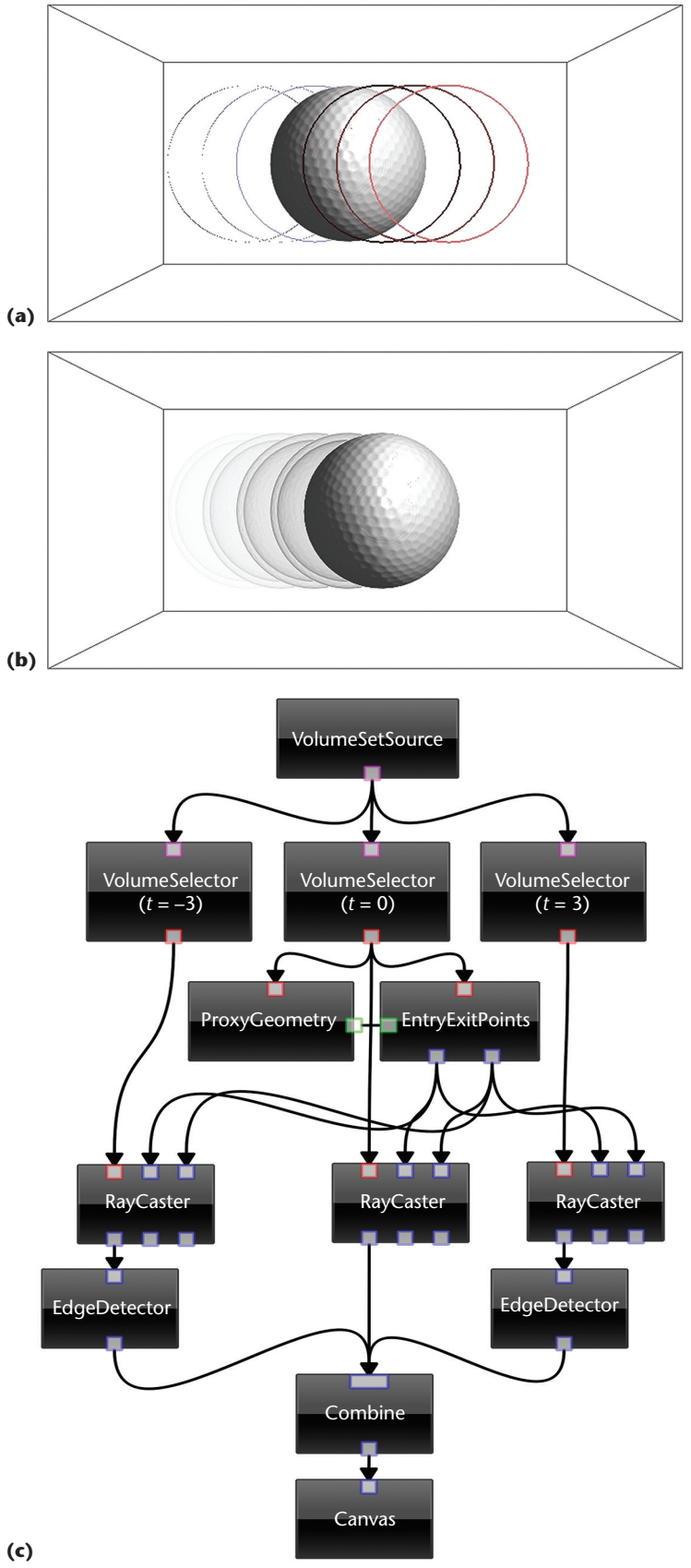


Figure 6. Two ways to visualize motion in a single still image. The volumetric data set contains a moving golf ball; we visualize its motion using (a) edges and (b) semitransparency. (c) The network for edge-based visualization is simplified for brevity.

network diagram; the actual network contains three `RayCasters` and `EdgeDetectors` each for the preceding and subsequent time steps.) `Combine` combines the resulting images in the proper order so that newer silhouettes cover older ones.

The second technique uses semitransparency to show preceding 3D volumes (see Figure 6b). It uses `RayCasters` to render the volumes, attenuates the volumes depending on their point in time, and combines the resulting images.

Our third technique uses speedlines, a technique that cartoons use to depict past motion, to visualize the movement of a simulation of hurricane Isabel (see Figure 7). A preprocessing step calculates each voxel's motion between two time steps. The network first uses a ray-casting sequence (`ProxyGeometry`, `EntryExitPoints`, and `RayCaster`) to produce a volume rendering of the rain's intensity (taken from the right `VolumeSelector`). The input for `SpeedlineRenderer` is the motion information derived from the rain's intensity (taken from the left `VolumeSelector`). This technique uses the collected motion vectors to calculate the speedlines' positions and directions. `VectorfieldRenderer` visualizes motion in the hurricane. `EdgeDetector` processors configured to produce a halo effect process all three output images. Furthermore, `TextureRenderer` loads and renders a texture containing a geographic map, and `Combine` combines all the images into the final visualization.

By splitting a complex ray-casting process into different tasks performed on different processors, Voreen provides a lot of flexibility because users can intervene at different points during ray casting. Voreen's OO design lets users easily create customized processor classes that cooperate seamlessly with existing classes. A user-friendly GUI supports rapid prototyping of visualization ideas.

We've implemented several applications based on our library—for example, for specific tasks in routine medical diagnostics. One example in this area is a system for diagnosing coronary artery disease based on SPECT data, while another one allows multimodal vessel inspection based on PET/CT (positron emission tomography/computed tomography) data. Voreen's source code, the networks we described, and sample data sets are freely available at www.voreen.org. In the future, we'd like to further extend Voreen's capabilities to make visualization prototyping even easier on all abstraction levels. Thus, we plan to realize a set of dedicated processor skeletons, which are solely configured through shader programs and can thus be modified at runtime.

Acknowledgments

This work was partly supported by grants from the German Research Foundation (DFG), SFB 656 MoBil (projects Z1, PM5). The hurricane Isabel data set is courtesy of the US National Center for Atmospheric Research.

References

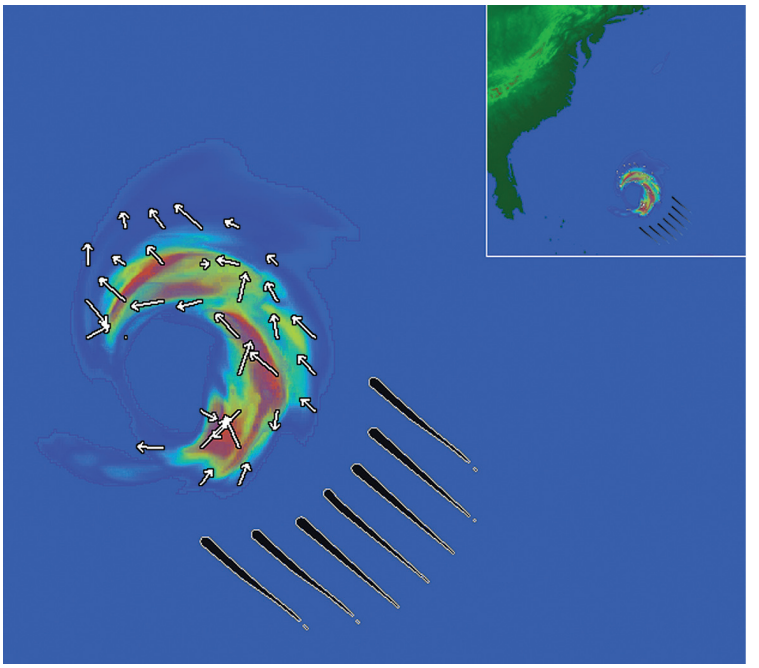
1. J. Krüger and R. Westermann, "Acceleration Techniques for GPU-Based Volume Rendering," *Proc. IEEE Visualization*, IEEE CS Press, 2003, pp. 287–292.
2. T. Ropinski et al., "Internal Labels as Shape Cues for Medical Illustration," *Proc. 12th Int'l Fall Workshop Vision, Modeling, and Visualization* (VMV 07), Akademische Verlagsgesellschaft AKA, 2007, pp. 203–212.
3. J. Meyer-Spradow et al., "Glyph-Based SPECT Visualization for the Diagnosis of Coronary Artery Disease," *IEEE Trans. Visualization and Computer Graphics*, vol. 14, no. 6, 2008, pp. 1499–1506.
4. J. Meyer-Spradow et al., "Illustrating Dynamics of Time-Varying Volume Datasets in Static Images," *Proc. 11th Int'l Fall Workshop Vision, Modeling, and Visualization* (VMV 06), Akademische Verlagsgesellschaft AKA, 2006, pp. 333–340.

Jennis Meyer-Spradow *is a senior software engineer in the Amira development team at Visage Imaging. His research interests include medical visualization, volume rendering, and GPU programming. Meyer-Spradow has a PhD in computer science from the University of Münster. Contact him at jmeyer-spradow@visageimaging.com.*

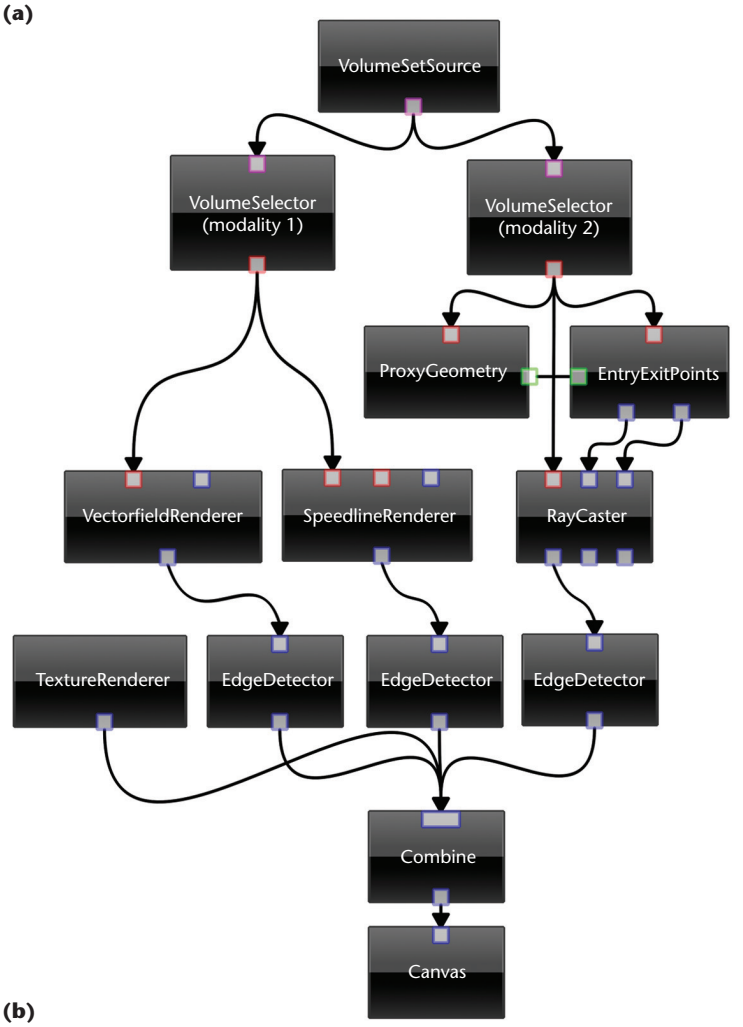
Timo Ropinski *is a senior research fellow in the University of Münster's Visualization and Computer Graphics Research Group. His research interests include scientific visualization and computer graphics techniques, with a special emphasis on volume rendering. Ropinski has a PhD in computer science from the University of Münster. Contact him at ropinski@math.uni-muenster.de.*

Jörg Mensmann *is a research associate in the University of Münster's Visualization and Computer Graphics Research Group. His research interests include volume visualization, efficient rendering algorithms, and GPU-based techniques. Mensmann has a Diplom (Dipl.-Inform.) in computer science from the University of Münster. Contact him at mensmann@uni-muenster.de.*

Klaus Hinrichs *is head of the University of Münster's Visualization and Computer Graphics Research Group. His research interests include visualization, computer graphics, and human-computer interaction. Hinrichs has a PhD in computer science from the Swiss Federal Institute of Technology (ETH) Zurich. Contact him at khh@uni-muenster.de.*



(a)



(b)

Figure 7. Using speedlines to visualize motion in still images. (a) Visualization of motion in a hurricane Isabel data set. (b) The network uses volume rendering to visualize the rain's intensity, speedlines to depict the overall motion, and arrows to show motion in the hurricane.