Vision, Modeling, and Visualization (2011)
Peter Eisert, Konrad Polthier, and Joachim Hornegger (Eds.)

Seamless Integration of Multimodal Shader Compositing into a Flexible Ray Casting Pipeline

S. Arens, M. Bolte and G. Domik

Department of Computer Science, University of Paderborn, Germany

Abstract

In the last three years a number of multi-volume GPU ray casting systems have been presented. Some of them are very powerful and provide a wide variety of features. However, these approaches are either only capable of displaying multiple modalities together without logically combining them or they lack the necessary flexibility for rapid visual development. These features are fundamental for visualizing the coherent information of multimodal data. In this paper we therefore present an integrated way of visually specifying a volume rendering pipeline including a flexible multimodal compositing of sampling, transfer functions, logical operators and shading. As a result the data flow can be visually constructed and retraced from preprocessing through to the shader operations. Hence intuitive visual prototyping of multimodal transfer function compositing is possible at runtime.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Multimodal data acquisition appears to be the most promising method for diagnosis by means of volumetric datasets in future. However, visualization of multimodal data is still quite complicated. Especially when modalities exhibit diverging resolutions, extents or orientations (e.g. they were taken from different scanners) their combined visualization is complex. While alternative visualization methods like glyphs (e.g. [OHG*08]) or "linked feature display" are already well investigated, up until a couple of years ago it has technically not been possible to visualize multimodal data via direct volume rendering, due to lack of computation power. Instead, compositing could only be applied in preprocessing, utilizing resampling of the volumes to a common multimodal grid, which introduces a lot of sampling errors and is time-costly. Therefore flexible experimentation with multimodal compositing was not imaginable. In the last few years there has been a great deal of work on GPU-based rendering methods for multiple overlapping volumes. However, for experimenting with multimodal visualizations the intersection of the volumes is most important, reducing the complexity of this problem. However, it is not only important to display several volumes, but also to combine their provided information in order to attain a more expressive visualiza-

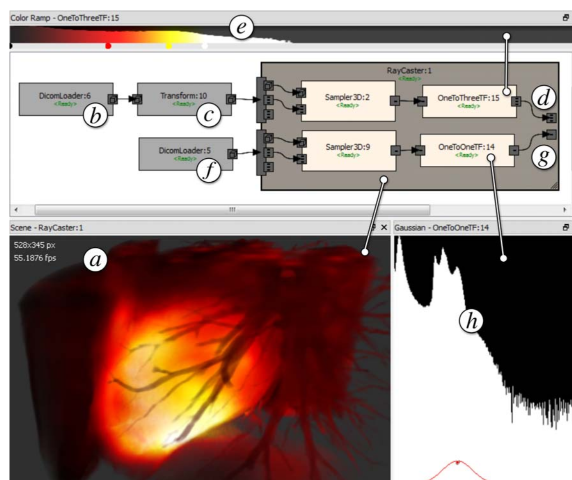

Figure 1: *Simple example of a multimodal pipeline in our system showing a heart with a ray casting renderer (a). The PET volume (b) is co-registered to the CT volume using a transformation node (c) and determines the rgb-color (d) for each sample, that is defined by the color ramp TF (e). The CT volume (f) on the other hand serves as the basis for the α value (g) of each sample, defined by a 1D TF (h). The three connection lines indicate which node provides which widget.*

© The Eurographics Association 2011.

DOI: 10.2312/PE/VMV/VMV11/347-352

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

tion. Most of the proposed systems are capable of displaying several volumes together, but not to combine their information. In this paper we present a flexible but integrated data flow, for exploring multimodal compositing schemes in direct volume rendering. Using this, the data flow can be easily constructed and retraced from preprocessing to the final composed color. This approach is aimed at engineers that experiment with new modalities and visualization methods. We provide an intuitive and simple interface, as we treat volumes, computed metrics (e.g. curvature or size) and segmentations uniformly within the pipeline. On the other hand this approach supports a deep understanding of the created visualizations, which is necessary to improve them. Hence, new logical combinations of modalities, metrics, transfer functions and segmentations can be created in an efficient way but without programming skills.

2. Related Work

Most of the recent papers dealing with multimodal volume rendering focus on rendering multiple arbitrary aligned volumes correctly in real-time, including all overlapping parts. There are essentially two strategies for solving this problem for GPU-based approaches: depth peeling [RBE08, PHF07, BBPtHR08] and binary space partitioning [LLHY09, LF09]. If someone wants to combine the information of different volumes logically, he needs to specify separate shading pipelines for all overlapping parts. E.g. in Figure 1, where PET is used for color and CT for opacity, three specifications would be necessary. One for the region with PET and CT, one for PET only and one for CT only; that is $2^n - 1$ specifications for n volumes, leading to a very high effort. But in medical diagnoses the information of different modalities (like PET and CT) needs to be combined, to be insightful. Since the organ of interest is always in all of these modalities, typically the specification of the region intersecting with all modalities is the most important one. That is why we decided to restrict the rendering to this region, demanding only one specification from the user. Nevertheless, this technique is extendable to other not completely overlapping parts of the scene.

In [CS99] three intermixing strategies have been formulated (Illumination Level Intermixing, Accumulation Level Intermixing and Image Level Intermixing), that were implemented in most of the stated publications. But these strategies are formulated in such a way, that a single volume is always given a visual appearance *unaffected* by the other volumes. The intermixing strategy only enables the joint display of the volumes with their individual appearance. This is because only *chains* of processing stages are defined, whose color values are joined by the selected intermixing scheme. *Graphs* are required in order to make use of the full information given at a sample's location in multimodal data by giving a sample a visual appearance *affected* by the other volumes. Figure 2 shows an example with three modali-

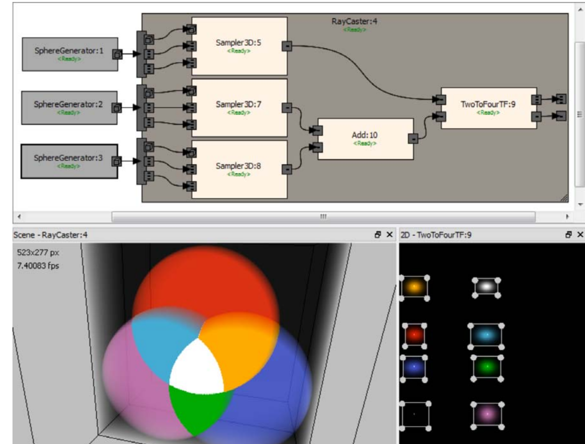

Figure 2: *All subsets of three overlapping volumes containing one different positioned sphere each are separated and colored individually during rendering using logical operators and a two-dimensional transfer function. The appearance of each modality is affected by all others.*

ties, where each modality is given an appearance affected by all others. However, in [BBPtHR08] two modalities can be combined by a two-dimensional transfer function to use their coherent information. Up to now, a general *graph* can only be constructed by the software system of [PHF07]; the other proposed systems only allow for data flow *chains*, that are merged by predefined intermixing schemes. Compared to our pipeline, the system in [PHF07] only allows for a specification of shading but is missing sampling that is important for seamless integration of segmentations and e.g. iso-surface rendering. Furthermore, their system lacks the direct visual connection between incoming data of the preprocessing and the shader graph, leading to a data flow that is difficult to retrace. Finally their system is using texture slicing whereas our system uses GPU ray casting.
In [RBE08, PHF07, LLHY09, LF09] shader instantiation is used to avoid costly conditional branching on the GPU. We adopted this technique and adapted it to the conditions of the flexible pipeline.

3. Data Flow Pipeline

Many volume rendering systems try to simplify the user interface as much as possible, regardless of technical details and properties. Such a concept is only adequate if the behavior of the software is always the same. In case of a modifiable and thus unsteady pipeline, it is necessary to visualize the behavior of the pipeline e.g. with a data flow graph. But the degree of freedom when constructing a pipeline is bounded, due to the fact that it needs to be technically realizable. I.e. global functions operating on volumes in the preprocessing and local functions operating on a per sample basis, that need to be performed on the graphics hardware,

© The Eurographics Association 2011.

dictate a rough framework to the structure of the pipeline. To control the data flow from loading to final coloring, we decided to construct it graphically in one single pipeline, despite of the named technical issues. It consists of elements with input and output ports, elements containing elements and connections that define the data flow between compatible ports. To make clear and enforce the technically given order of preprocessing functions and local functions per sample, these element groups are visually *nested*. Outside the rendering element is the data flow of the preprocessing stages. Inside the rendering element is a subpipeline that produces a color and an opacity for each sampling position. This transition is thereby visible and enforced but unobtrusive concerning the interaction with the user interface.

The shading elements of the subpipeline provide GLSL snippets that are composed to a shader program, that is used for the GPU-based rendering. The connection between the outer and inner data flow is specified clearly by connection ports. Small icons at the ports and more detailed tooltips reveal the signature of a port, and hence, which elements can be connected. All elements and connections are inserted via drag&drop into the data flow graph, whereby connections between incompatible ports and circles within the pipeline are automatically suppressed. Figure 2 shows a result illustrating the flexibility of our system, which provides the full power of set theory.

The pipeline concept fits to all renderers that provide sampling positions. As most other renderers do that use ray casting, we encode the start and end points of each ray in precomputed entry and exit textures. We currently provide a ray caster and a renderer for curved planar reformations [KFW*02]. Both use the same render core and only distinguish in the way they compute the rays' entry and exit positions. Hence, both encapsulate the same concept for constructing subpipelines to generate shader code. We decided to not pass entry and exit textures as parameters through the pipeline, since this part of the pipeline is almost always the same. Hence, renderers must provide their own entry/exit texture computation. But that clearly simplifies the pipeline specification.

3.1. Elements

Although being technically severely different, the user handling of the preprocessing elements and the shading elements are absolutely identical. This way the user is not confronted with the difference of shaders, CPU programs or GPGPU programs. As a second advantage, the whole data flow can be visualized in a single graph, leading to a much easier creation and specification of the data flow. Elements that contain inner pipelines can be displayed "expanded" and "collapsed" to improve the overview.

The result of many elements does not only depend on the input values but on additional *properties*. These properties

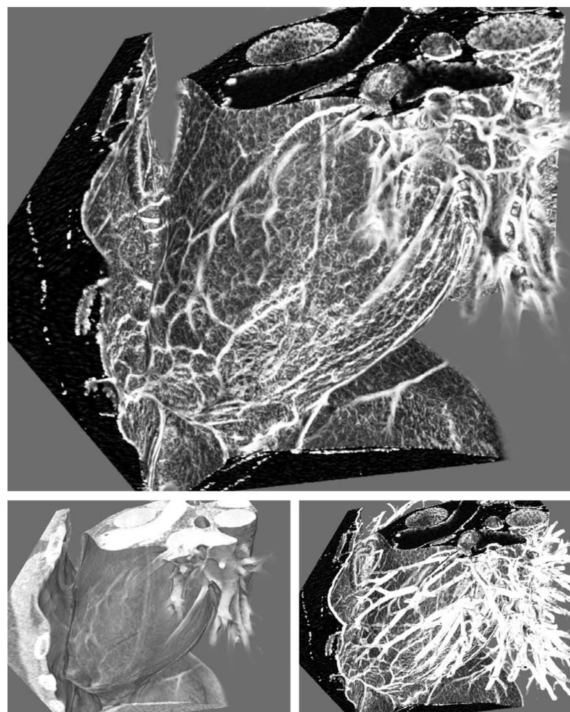

Figure 3: *Example of how different metrics computed in preprocessing can serve special visualizations. Shown are vessels on the pericard in a CT-angiographic dataset enhanced by curvature and gaussian metrics. This kind of combination is only possible with data flow graphs. Figure 4 shows the corresponding pipeline. The small pictures underneath show results without the use of curvature (left) and without suppressed lungs (right). Even those would not be possible with data flow chains.*

are displayed on a property pane that comes with every element. See Figures 5 – 7 for an exemplary shader element that uses properties. To develop new elements more rapidly, we programmed a generator that produces dummy elements (preprocessing elements as well as shader elements) with ports and properties, that are automatically arranged on the property pane and saved, when the pipeline is being saved. This way the programming of new elements is reduced to a minimum. E.g. for most shading elements only the body of one GLSL function has to be programmed.

3.2. Preprocessing Pipeline

In the preprocessing part of the pipeline data is loaded and processed by algorithms that need to run only once (or at least not every frame) and do not affect the shader program. The processed data are basically volumes, meshes and points and are passed through by the connections. Volumes are passed through voxel by voxel instead of moving whole buffers. This way memory consumption can be reduced to

© The Eurographics Association 2011.

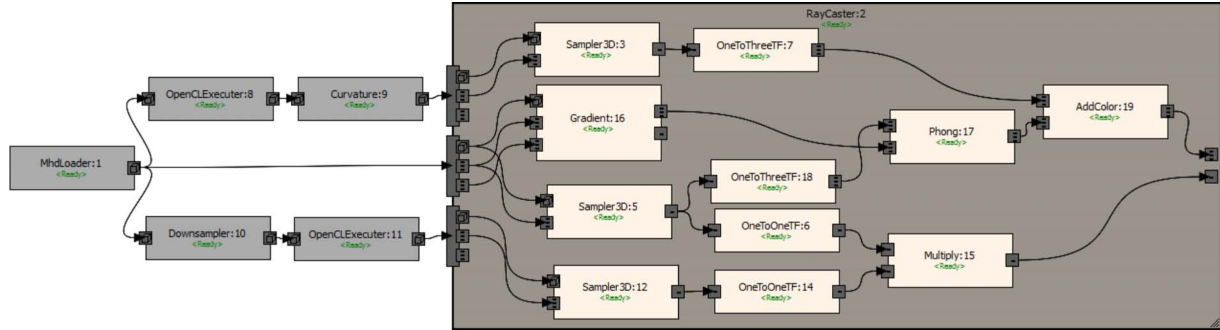

Figure 4: *Pipeline of Figure 3. In the upper chain of the pipeline the CT data is preprocessed by a gaussian filter followed by a computation of the curvature κ_1 to emphasize the small vessels of the pericard in the original dataset. The lower chain is used to suppress the lung vessels by eroding them with a widespread gaussian filter. The ray casting element contains 3 samplers, 4 transfer functions, a gradient element, a phong element, a multiply- and an add-element to combine the metrics usefully. The whole data flow from preprocessing to final shading can be retraced in one graph.*

a minimum, because many functions can be implemented inline without caching (e.g. an element that crops the incoming volume can be implemented by just modifying the requested index). Ports provide metadata to hold resolution, voxel spacing, transformation etc.. In contrast to software systems that use image level intermixing, in this stage there is no data flow containing images or other rendering data.

Some of our preprocessing elements make use of OpenCL, to reduce execution time. Additionally we decided to provide a generic OpenCL element that processes OpenCL programs that contain simple convolutions. Besides these simple convolution filters we are currently able to compute more complex metrics such as median, standard deviation, size [CM08], vesselness according to [PBB05] and [FNVV98] and curvature [KWTM03] to use them as additional modality.

Figure 3 shows how metrics computed in the preprocessing can visualize the individual vessels on the pericard while simultaneously suppressing the lung vessels (that share the same value and curvature). With data flow *chains* only, these visualizations would not be possible. Figure 4 shows the corresponding pipeline. The two gaussian filters make use of the stated generic OpenCL element.

3.3. Shading Pipeline

The shading pipeline is always a subpipeline of a renderer. It contains shading elements that provide GLSL shader functions. Due to the varying elements, the varying number of volumes and meshes used, the shader is generated during runtime by the *shader composer*. This way costly conditional branches in the shader are avoided. Input and output ports are directly made available as GLSL variables and properties are automatically bound to uniforms. To avoid naming collisions between multiple instances of the same shading element the variables are given a unique name internally.

The position of the samples depends on the renderer. Therefore the renderer passes the volumes including the current sampling position as well as the texel spacing to the inner shading pipeline. This *integrated embedding* is also represented by a triple-port that connects data flow between inner and outer pipeline. The expected result of each sampling procedure is an RGB and an opacity value, which is why these two ports are placed on the right side of each renderer.

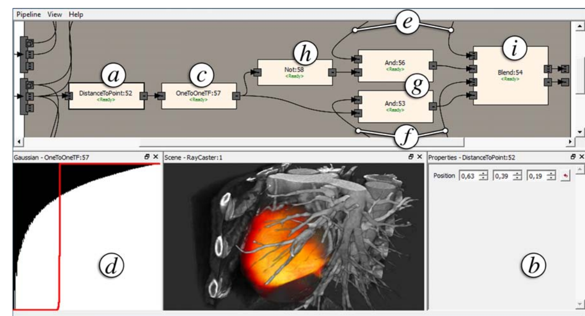

Figure 5: *The upper part of the figure shows a portion of an inner shading pipeline. The element "DistanceToPoint" (a) computes the sample's distance to a point specified in the properties (b). Based on this metric the transfer function (c) defines a function (d) that distinguishes near and far samples. In the not shown part of the pipeline two appearances (i.e. color and opacity per sample) are defined, one for the near region (e) and one for the far region (f). The "And" elements (g) in combination with the "Not" element (h) use the mapping of the "distance transfer function" to set the alpha value of one appearance to zero before blending (i). See Figure 6 and 7 for C++ and GLSL shader code used for the "DistanceToPoint" element.*

© The Eurographics Association 2011.

```
/** This shading element computes the sampling position's distance to the specified point. */
DistanceToPoint::DistanceToPoint() :
    ShaderElement("DistanceToPoint"),
    _samplingPosInputPort(new Port(this, "vec3", "samplingPos", Port::Direction_Input)), // create input port
    _distanceOutputPort(new Port(this, "float", "distance", Port::Direction_Output)),    // create output port
    _pointProperty(new Vector3Property("point", "Point",                                 // create property, id, label
                                       "Point to compute distance to", this,             // description
                                       vec3(0.5f), vec3(0.0f), vec3(1.0f), vec3(0.01f))) // default, min, max, step
{
    addPort(_samplingPosInputPort);                                                      // add ports
    addPort(_distanceOutputPort);
    addProperty(_pointProperty);                                                         // add and connect property
    connect(_pointProperty, SIGNAL(valueChanged()), this, SIGNAL(shaderOutputChanged()));
    addShaderUniform("point", "vec3", _pointProperty);                                   // add uniform "point" and
}                                                                                        // bind to _pointProperty
```

Figure 6: *C++ code for constructing the "DistanceToPoint" element (Fig. 5 (a)) and its properties (Fig. 5 (b)). Only a few lines have to be modified when creating new elements with the element generator. Input and output ports are directly made available as GLSL variables. Properties are bound using addShaderUniform. See Figure 7 for the corresponding shader code.*

```
distance = length(point - samplingPos);
```

Figure 7: *The complete GLSL shader code that has to be written for the "DistanceToPoint" element. The variables samplingPos, distance and point are bound to their corresponding ports and properties by the shader composer. See Figure 6 for the corresponding C++ code.*

Within the pipeline all compatible ports can be connected. This way e.g. the gradient can serve as color. We decided to consequently disjoin color and opacity values in the rendering pipeline. This leads to a better understanding compared to joined RGBA values and to a faster specification compared to completely disjoined red, green, blue and alpha channels. Nevertheless, there is no limitation that prohibits to pass other data types than "float" and "vec3" through the ports of the elements.

For *logical combination* of the modalities, there currently exist the shader elements AND, OR, XOR, NOT, ADD, ADD-Color, Multiply, Multiply-Color and Blend-RGBA. Each of them combines the values at the input ports using the corresponding logical operator. Additional operators can be generated within seconds, using our element generator. Additionally two-dimensional transfer functions can be used to map any two floats within the pipeline to one value.
All in all, for the mapping of values we provide a one-dimensional, a two-dimensional and a style-based *transfer function*. Each of them can output a three-dimensional vector (typically used as color), a floating point value (typically used as alpha), or both. Using the logical operators an arbitrary-dimensional transfer function can be constructed.

Figures 6 and 7 show the main C++ and GLSL code that is mandatory for a shader element that computes the sample's distance to a specified point in the properties. Figure 5 shows how a transfer function and logical operators can be used to merge two appearances dependent on the result of this element. Since the distance is computed in the shader, the sphere with colored appearance can be moved through the volume in realtime by changing the position of the specified point in the properties.

3.4. Texture Lookup

The texture lookup is done within *sampling* elements in the pipeline. This way sampling options, such as nearest neighbor, trilinear interpolation and iso-value, can be chosen independently from the input volume. Additionally, segmentations can be processed by the same pipeline elements as used for other modalities by choosing nearest neighbor sampling. For example it is possible to use a transfer function that maps the selected segmentation label to 1 and all other labels to 0.3. Multiplying the result to alpha (using the multiply element) fades out everything outside the selected segmentation. Figure 8 demonstrates this pipeline inside a curved pla-

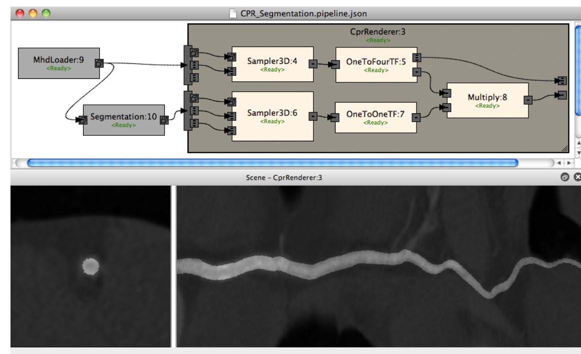

Figure 8: *Segmentations integrate seamlessly into the pipeline and are simply treated as another modality. Here a segmentation is used to modify the opacity value in a curved planar reformation of a coronary artery.*

© The Eurographics Association 2011.

nar reformation renderer. Notice that the sampler element for the segmentation needs the voxel spacing as a third parameter to take samples via nearest neighbor method.

3.5. Histograms

At each position of the shading pipeline, 1D and 2D *histograms* can be computed, that accumulate the values up to this stage in the pipeline. Figure 9 shows a simple example of histograms taken at different stages of the pipeline. This is a particularly useful feature when combining several metrics and modalities by logical operators. Every transfer function makes use of this feature, too. The computation of the histograms is done with the method of [SH07] on the GPU and was extended to 2D histograms. The computation takes about 100ms, leading to an almost real-time update when changing a transfer function. To achieve this speed we reduced the number of samples to 5 million, which reduces the quality of the histogram hardly noticeably.

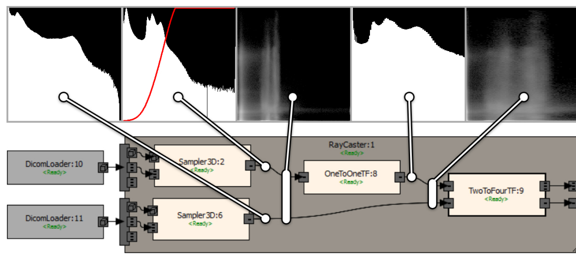

Figure 9: *Five histograms taken at several stages of the pipeline. The first three histograms show the original data in 1D and 2D histograms. The fourth and the fifth show the histograms after manipulation by a 1D transfer function as shown by the red line.*

4. Conclusion

In this paper we presented a volume rendering software system, which allows a construction of arbitrary multi-volume pipelines. Volumes in form of original data or modalities, metrics and segmentation can be arbitrarily combined using logical operators rather than simply displaying them together. This way the advantages of the evolving imaging technologies can be combined without any programming. As far as we know, our system is the first one that is capable of this combination, using ray casting. Furthermore, our concept is the first one that combines the visualization of the preprocessing's data flow with that of the GPU rendering seamlessly. In future we want to make the graph more self-explanatory and add links between elements that shall share the same properties or parameters. Furthermore, we are interested in porting this pipeline concept to an open source volume rendering platform to make it available to a wider community.

Acknowledgement

We appreciate cooperation with Dr. W. Burchert, R. Weise, J. Holzinger and H. Fricke of the Heart and Diabetes Center North-Rhine Westphalia, Germany. We deeply appreciate Erik Bonner for his help, suggestions and review of this work.

References

[BBPtHR08] Brecheisen R., Bartroli A. V., Platel B., ter Haar Romeny B. M.: Flexible gpu-based multi-volume ray-casting. In *VMV* (2008), Deussen O., Keim D. A., Saupe D., (Eds.), Aka GmbH, pp. 303–312.

[CM08] Correa C. D., Ma K.-L.: Size-based transfer functions: A new volume exploration technique. *IEEE Trans. Vis. Comput. Graph. 14*, 6 (2008), 1380–1387.

[CS99] Cai W., Sakas G.: Data intermixing and multi-volume rendering. *Computer Graphics Forum 18* (September 1999), 359–368(12).

[FNVV98] Frangi A., Niessen W., Vincken K., Viergever M.: Multiscale vessel enhancement filtering. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI'98*, Wells W., Colchester A., Delp S., (Eds.), vol. 1496 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998, pp. 130–137.

[KFW*02] Kanitsar A., Fleischmann D., Wegenkittl R., Felkel P., Gröller E.: Cpr - curved planar reformation. In *IEEE Visualization* (2002).

[KWTM03] Kindlmann G., Whitaker R., Tasdizen T., Moller T.: Curvature-based transfer functions for direct volume rendering: Methods and applications. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 67.

[LF09] Lux C., Fröhlich B.: Gpu-based ray casting of multiple multi-resolution volume datasets. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II* (Berlin, Heidelberg, 2009), ISVC '09, Springer-Verlag, pp. 104–116.

[LLHY09] Lindholm S., Ljung P., Hadwiger M., Ynnerman A.: Fused multi-volume dvr using binary space partitioning. *Comput. Graph. Forum 28*, 3 (2009), 847–854.

[OHG*08] Oeltze S., Hennemuth A., Glasser S., Kühnel C., Preim B.: Glyph-Based Visualization of Myocardial Perfusion Data and Enhancement with Contractility and Viability Information. In *VCBM* (2008), pp. 11–20.

[PBB05] Pock T., Beichel R., Bischof H.: A novel robust tube detection filter for 3d centerline extraction. In *18th Symposium on Operating System Principles (SOSP* (2005), Springer-Verlag, pp. 481–490.

[PHF07] Plate J., Holtkaemper T., Froehlich B.: A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics 13* (November 2007), 1584–1591.

[RBE08] Roessler F., Botchen R. P., Ertl T.: Dynamic shader generation for flexible multi-volume visualization. In *Proceedings of IEEE Pacific Visualization Symposium* (2008), pp. 17–24.

[SH07] Scheuermann T., Hensley J.: Efficient histogram generation using scattering on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 33–37.

© The Eurographics Association 2011.