DOI: 10.1111/j.1467-8659.2011.01952.x

Eurographics / IEEE Symposium on Visualization 2011 (EuroVis 2011)
H. Hauser, H. Pfister, and J. J. van Wijk
(Guest Editors)

Volume 30 (2011), Number 3

A Shader Framework for Rapid Prototyping of GPU-Based Volume Rendering

Christian Rieder[1], Stephan Palmer[2], Florian Link[2] and Horst K. Hahn[1]

[1]Fraunhofer MEVIS, Germany
[2]MeVis MEDICAL SOLUTIONS AG, Germany

Abstract

In this paper, we present a rapid prototyping framework for GPU-based volume rendering. Therefore, we propose a dynamic shader pipeline based on the SuperShader concept and illustrate the design decisions. Also, important requirements for the development of our system are presented. In our approach, we break down the rendering shader into areas containing code for different computations, which are defined as freely combinable, modularized shader blocks. Hence, high-level changes of the rendering configuration result in the implicit modification of the underlying shader pipeline. Furthermore, the prototyping system allows inserting custom shader code between shader blocks of the pipeline at run-time. A suitable user interface is available within the prototyping environment to allow intuitive modification of the shader pipeline. Thus, appropriate solutions for visualization problems can be interactively developed. We demonstrate the usage and the usefulness of our framework with implementations of dynamic rendering effects for medical applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Interaction Techniques I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms I.3.6 [Computer Graphics]: Methodology and Techniques—Dynamic Shader Generation

1. Introduction

Direct volume rendering (DVR) is used to create meaningful images from 3D data sets. GPU-based approaches that allow for interactive rendering on consumer graphics hardware have been proposed. Interactive GPU-based DVR is used in many fields, particularly in medical visualization. However, the creation of meaningful visualizations for medical diagnosis is challenging, due to the particular needs of the medical users [MLZ*02]. In the literature, effective visualization algorithms have been proposed, which often focusing on specific scanning strategies and medical questions [BHWB07]. A key problem within common rendering frameworks is the high amount of effort required to develop appropriate visualizations and to customize the algorithms rapidly.

In this work, we present a rapid prototyping framework for GPU-based volume rendering, which supports computer scientists in the development of volume visualizations. Our framework extends the volume renderer of *MeVisLab* [MeV11], a freely available development environment for medical image processing and visualization, to allow the developer to interactively append custom shader code. The main contributions of our paper are:

- The determination of important requirements for the development of a rapid prototyping framework for volume rendering upon which our design decisions are based.
- The introduction of a dynamic rendering configuration to facilitate a customized extension of the rendering shader. Technically, we present a flexible and modularized shader pipeline based on the *SuperShader* concept.
- The realization of a rapid prototyping environment, in which custom shader code can be interactively edited and attached to the shader pipeline during run-time.
- The demonstration of the usage and usefulness of our system, which has been used to rapidly develop medical visualizations and integrate them in clinical applications.

2. Related Work

Dynamic shader creation has been a research topic even before modern programmable GPUs became available. It is an important topic in many fields of image generation, not just

volume rendering. One of the first approaches for the combination of multiple techniques for illumination and texturing are *Shade Trees* [Coo84], which combine basic shading operations using a data flow concept. *Building Block Shaders* by Abram and Whitted [AW90] combine that approach with a graphical user interface to enable efficient generation of shader programs and represent the first *Visual Shading Language*. In recent years, similar implementations have been proposed for GPU-based realtime rendering [GBD04, MSPK06, ME09]. All of these visual programming based approaches are aimed at simplifying the shader definition for the user, e.g., for an artist [Tat04, GD06]. However, this does not reflect the needs of application-controlled shader adaption for GPU-based realtime rendering in complex visualization systems or games. Such scenarios require the generation of specific effects at runtime without needing to store all possible permutations [FW04].

A concept designed for these special needs is the *SuperShader* [McG05]. The basic idea of this approach is to break down the whole shader into smaller parts [Har04], such as its subdivision based on components of the underlying rendering pipeline. The SuperShader contains all of the potentially required parts in the form of code snippets and allows the deactivation of undesired parts. Communication between the parts is achieved by means of a global data structure. Trapp et al. [TD07] extend that approach by adding a *Shader Management System* that concatenates the shader at runtime instead of disabling unused parts.

Stegmaier et al. [SSKE05] present a simple and flexible volume rendering framework for GPU-based raycasting. New algorithms are implemented by writing completely new shaders. Bruckner et al. [BG05] present *VolumeShop*, a prototyping platform for visualization research, and direct volume illustration in particular. Its main aim is to provide maximum flexibility to the developer to create illustrative visualizations such as exploded views or three-dimensional selection painting. Also, several volume rendering frameworks specifically address rendering of multiple intersecting volumes. Brecheisen et al. [BiBPtHR08] describe a raycasting-based multi-volume rendering system, which uses a depth-peeling approach to combine geometry and multiple volumes in one rendering. Plate et al. [PHF07], present a multi-volume shader framework that focuses on rendering very large, multi-resolution intersecting volumes. A shader composer which is based on the data flow model allows composing predefined shader nodes to a final rendering shader at run-time. *Voreen* [MSRMH09], based on raycasting, provides a graphical user interface for defining a network of so-called processors which together assemble a render pipeline. Predefined functions of a shader library can be edited or substituted by the user, which are included into the final shader. Although *Voreen* gives control over the rendering pipeline, it does not allow dynamically changing parts of the rendering shader without modifying the remaining shader code. This is possible within the render graph concept introduced by Rössler et al. [RBE08a, RBE08b]. Using this approach, modularized shading algorithms can be defined and interactively combined to render multiple volumes. To extend this system with new render nodes, one inherits from existing C++ base classes. Thus, the abstract render nodes can not be edited on the fly, nor can they automatically created by the application. In contrast to these concepts, we propose a framework which focuses on the developer- or application-controlled dynamic editing of modularized shader code.

Bitter et al. [BVUW*07] compare four freely available frameworks for image processing and visualization that use ITK. A survey of the most successful open-source libraries and prototyping frameworks for medical application development is presented in [CJN07]. Another prototyping environment is the eXtensible Imaging Platform (XIP) [PET07, XIP11], which has basically a comparable functionality as *MeVisLab*. To our knowledge, none of the related prototyping and volume rendering frameworks allow dynamically extending the rendering shader with modularized custom shaders snippets which can be freely edited by the developer during run-time.

3. Rapid Prototyping

Our motivation to use rapid prototyping for volume rendering is the possibility of developing appropriate solutions for a current visualization problem in an interactive way, without recompilation of the C++ program and to quickly integrate the solution into stand-alone applications. For instance, in medical visualization, specific visualizations have to be created for image-guided surgery planning, diagnosis support for various diseases, or anatomical exploration of the patient data. Often, changing high-level rendering parameters of existing features does not solve such complex visualization problems. Hence, the volume renderer has to be extended with new features, which is often not possible in the underlying C++ code level at run-time. Because this strategy runs counter to the concept of rapid prototyping, the renderer has to be configured in a level between low-level C++ code programming and high-level parameter adjustment. Thus, the following are important requirements for DVR-based rapid prototyping:

Rendering Extension. The developer has to be able to manipulate the rendering configuration, particularly the shader code. To create new rendering effects, modularized custom shader code should be created and positioned into the existing rendering configuration while ensuring a valid shader.

Dynamic Rendering Configuration. To allow the evaluation of the rendering extension at run-time of the prototyping application, a dynamic rendering configuration is needed. Hence, the shader code of the volume renderer has to be structured in such a way that high-level changes of the rendering configuration result in the implicit modification of the underlying shader code.

Function Library. In an efficient prototyping environment, often-used shader functionality should be defined in shader functions and stored in a function library. With access to these functions, the user is able to rapidly implement custom shader code.

Suitable user interface. To facilitate the implementation of the modifications mentioned above, a suitable user interface has to be available within the prototyping environment. Such a user interface allows creating custom shader code, which uses the function library and specifying the extension of the renderer with the custom shader. Additionally, shader parameters, such as uniform variables and texture samplers, can be attached to the renderer.

4. Dynamic Shader Pipeline

The dynamic shader pipeline is the core functionality, which allows interactive rendering extension and dynamic configuration of the volume rendering. We use the OpenGL Shading Language (GLSL).

4.1. Design Decisions

The design decisions upon which our system is based relate to three key ideas:

- Breaking down the shader code into areas containing code for different computations.
- Definition of shader code as freely combinable, modularized shader blocks.
- Insertion of custom shader code between blocks without invalidating the final shader.

We identify the *Shade Trees* as well as a *SuperShader*-based concept as possible approaches for our modularized framework. In the *Shade Tree* concept, modularized shading components are arranged as a tree. The shading result of every basic shading block is propagated up the tree until the final result in the root is reached. The root denotes the output, and the leaves denote the input of the shader. Generally, *Shade Trees* allow high flexibility and are well suited for visual representation, but the implementation is laborious.

The *SuperShader* concept assumes the existence of a finite number of fragments, ordered in a linear list. Every fragment contains a code snippet that calculates an effect. The effect shaders are generated and optimized by a control shader at run-time using static branching.

Originally, we decided to use the *Shade Trees* to implement a shader pipeline in a prototypical framework. Atomic shader blocks can be edited and connected using a visual representation of the graph. The replacement of shader blocks is possible considering the signature of input and output values, whereas the internal shader code can be neglected. After discussions and user evaluations, we decided to switch to the *SuperShader* concept, because replacing shader code under consideration of the child and parent node's signatures is difficult to handle in practice. To replace atomic shader blocks with custom functionality, the signature has to be well known. In contrast, the signatures of all shader fragments in the *SuperShader* approach are identical. Thus, the shader can be combined more freely, and custom shader code may inserted flexibly. Furthermore, we found the linearly ordered list of the shader blocks a more intuitive representation of the volume rendering pipeline.

4.2. Basic approach

In our *SuperShader*-based approach, we directly copy the snippets into the shader code instead of using a control shader, allowing the addition of custom shader code dynamically. We utilize two general ideas of the *SuperShader* concept:

1. The assumption that a shader can be subdivided into a linear order of rendering operations such as lighting or transformation.
2. The code snippets use a global data structure for communication which is passed to the snippets at execution. Input values are read from the data structure and output values are written into the data structure.

4.2.1. Shader Pipeline

We subdivide the rendering shader into a linearly ordered shader pipeline and model a pipeline step as a subset of several shader pipeline functions, which contain the GLSL shader code. Figure 1 shows a schematic overview of our rendering pipeline.

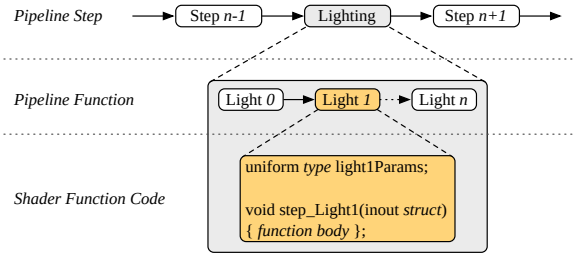


Figure 1: *Three layers of the rendering pipeline. Pipeline steps are in the top layer, which are subdivided into a subset of pipeline functions. The GLSL code itself is located below, in the shader function code layer.*

The following parameters have to be explicitly defined for each pipeline function for introspection: *shader parameters*, *struct parameters*, and *library functions*. Thus, several shader parameters, such as varying or uniform variables, can be attached to each pipeline function. Furthermore, the global data structure is a globally defined *struct* in the final shader. The struct is passed as input and output parameter to the pipeline functions. Input values are read from the

struct and output values are written into the struct. The struct can be interpreted as the global state of the pipeline which represents the intermediate and final results of the rendering computation. Thus, all pipeline functions have the same signature and no return value and can only communicate via the global struct. Figure 2 shows the shader code of a pipeline function, which writes shading multipliers into the global struct.

```
uniform vec4 lightColor0;
uniform vec3 lightVector0;
uniform vec3 halfVector0;
uniform float specularity;
```

```
void vrStep_directionalLight0(inout globalStruct state) {
  vec2 result = vrLib_getShadingFromLight(state.gradient,
                                          lightColor0,
                                          lightVector0,
                                          halfVector0,
                                          specularity);
  state.diffuse0 = result.x;
  state.specular0 = result.y;
}
```

Figure 2: *Pipeline function to obtain shading from a light source using a library function. The function reads the gradients from the global struct and writes both, diffuse and specular multipliers back into the struct.*

To allow the reuse of common shader components such as shading or texture fetching, we also propose the use of a function library. A library function is specified as a code snippet which calculates a specific shader effect or color value per sample. Because library functions are stored in an external library and are generally independent from the pipeline, global shader attributes can not be attached. Thus, library functions have no access to the global struct of the pipeline and have to write the calculated effect as a return value. Figure 3 shows the shader code of a library function which calculates the diffuse and specular multipliers for a light source.

```
vec2 vrLib_getShadingFromLight( in vec3 inGradient,
                                in vec4 inLightColor,
                                in vec3 inLightVector,
                                in vec3 inHalfVector,
                                in float inSpecularity) {
  float diff = max(dot(inGradient, inLightVector), 0.0);
  float spec = max(dot(inGradient, inHalfVector), 0.0);
  spec = inLightColor.a * pow(spec, inSpecularity);
  return vec2(diff, spec);
}
```

Figure 3: *The library function returns diffuse and specular multipliers as a vec2 required for volume shading.*

4.2.2. Shader Generation

To generate the final shader, a shader factory algorithm combines the resulting shader from the active pipeline steps. The shader pipeline consists of the *pipeline map*, a data structure to reference pipeline functions. The corresponding shader string is generated using the pipeline map.

Setup of Data Structure. The key of the pipeline map is *pipeline position*, and the value is *pipeline step*. A pipeline step contains an ordered list of pipeline function references. For all pipeline steps of the data structure, the active pipeline functions are added.

Generation of Shader String. Subsequently, the pipeline map is used to generate the corresponding shader string. Input is the ordered list of functions, given implicitly in the map. Output is the final shader string. The following steps are processed:

1. The used information (shader pipeline context) for each pipeline function is stored in an additional temporary data structure:
   - Used varyings/uniforms.
   - Used parameter struct entries.
   - Used library functions.
2. The declarations (stored in the temporary data structure) are appended to the shader string:
   - Varying/uniform declarations.
   - Parameter struct declaration.
   - Library function declarations of used functions only.
   - All pipeline function declarations.
3. The main function is appended to the shader string:
   - Struct instantiation of the default values for the struct.
   - Call to pipeline functions in correct order (as defined in the map).

Figure 4 (a) gives an illustrative overview of the resulting *SuperShader* pipeline.

4.3. Application to Volume Rendering

In our slicing-based volume rendering system [LKP06], we define 16 pipeline groups for the available rendering effects (see Figure 4 (b)). The pipeline steps are grouped into four main rendering steps according to the general volume rendering pipeline. The first group consists of the *Sampling* pipeline steps, which fetch the sampling value for each active data set. The *Classification* group contains the steps for the application of corresponding transfer functions to the samples. Shading operations such as lighting, boundary enhancement, or tone shading are located in the *Shading* group. The *Compositing* group is the last group in the pipeline, containing the final composition of the samples to calculate the output fragment color.

Due to input–output parameter dependencies of the pipeline functions (e.g., gradients are needed for illumination calculation), the positions of the pipeline steps are arranged in a fixed order. However, the shader code of the

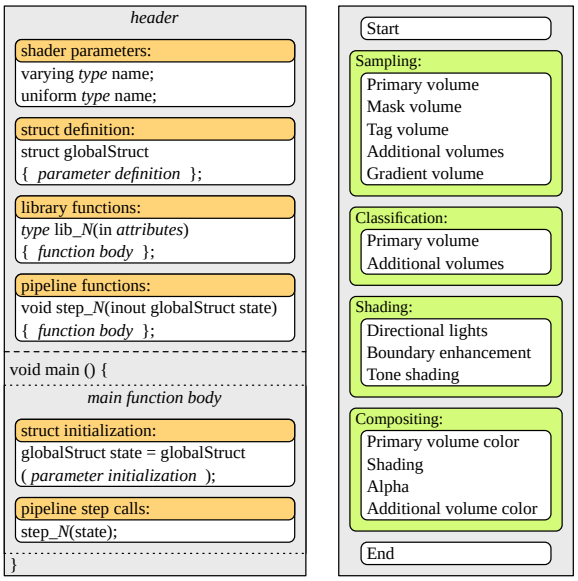


Figure 4: *(a) Overview of the composed SuperShader code of the pipeline with library and pipeline function. (b) The 16 pipeline steps are grouped in four main rendering steps.*

pipeline steps may be changed depending on rendering parameters, e.g., on the number of active light sources. Generally, the pipeline steps are activated according to the enabled rendering effects of the volume renderer. For instance, if volume shading is enabled, the rendering system has to activate the gradient calculation pipeline step, which is positioned before the illumination step. Because the implementation of shader functions are generally independent from each other, the underlying gradient calculation algorithm may be changed (texture based or on-the-fly [HKRs*06]) during run-time without affecting other pipeline steps.

4.4. Application to Slab Rendering

In addition to 3D renderings, we utilize our volume renderer to enable rendering orthographic projections, which allows efficient visualization of arbitrary multi-planar reformations. Furthermore, the data sets have to be loaded only once into memory and can be shared by multiple renderers. Hence, the shader pipeline can also be utilized for advanced 2D slice visualizations.

5. Prototyping Environment

The developed visual prototyping environment supports a Graphical User Interface (GUI) and scripting functionality to allow customization of the GLSL shader pipeline of the volume renderer. An API to develop C++ code is also supported.

5.1. Volume Rendering Scene Graph

Our volume rendering system is based on a scene graph. The volume renderer itself is a scene graph node. Supplementary nodes are various volume data nodes (e.g., additional volume and mask volume), transfer function nodes, and rendering effect nodes such as shading or boundary enhancement [ER00]. The scene graph concept allows the same volume renderer to be used in different scenes. Thus, the data set has to be loaded once into memory and can simultaneously be visualized in multiple viewers with varying render configurations.

If the state of a node changes, the scene graph will be traversed at the next frame and all node elements will be collected. Subsequently, the shader factory of the renderer generates the shader string and compares it with the last compiled shader program. If the shader programs differ from each other, the new shader is compiled and used for rendering. Due to the compilation process of the alternated shaders, a performance overhead can be observed. Thus, the compiled shaders are cached, allowing rapid reuse of previously executed shader configurations. Figure 5 illustrates the shader generation process.

Because of the visual representation of the nodes in our prototyping environment, the developer is able to connect the nodes to the scene graph interactively. Thus, the volume rendering can easily be extended at run-time.

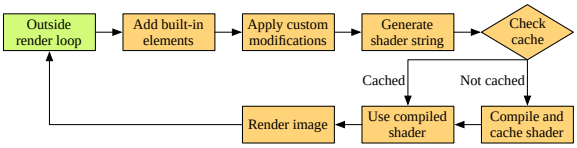


Figure 5: *Illustration of the shader generation process which has to be repeated for every frame.*

5.2. Custom Shader Pipeline

The presented fixed pipeline steps with the pipeline functions implement the available rendering effects in the *SuperShader*. To allow extension of the rendering pipeline with additional rendering effects, we introduce custom pipeline functions. A custom pipeline function is a shader function for which the function body as well as the parameters can be edited by the user. Custom pipeline functions can be attached to a pipeline step in order to add or replace existing pipeline step elements (see Figure 6). Therefore, we present the shader function node, the core functionality to edit the custom shader function and to control the appending to the pipeline. Two additional nodes allow adding shader parameters and attaching shader code to the header, such as library functions to the shader pipeline. Inspector nodes can be used to visualize the current pipeline as well as the composed shader used for rendering.

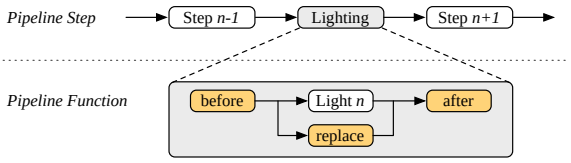


Figure 6: *Custom pipeline functions (orange) can be attached before or after, or replace the pipeline functions of a selected pipeline step.*

5.2.1. Shader Function Node

The shader function node is used to implement a custom pipeline function. We designed the GUI to be as simple as possible to allow fast and intuitive modifications without limiting the ability to create advanced shader effects.

Function Declaration. A text field with GLSL syntax highlighting allows the function body to be implemented and the function name to be defined (see Figure 7). Access to the global struct is achieved via the instance name *state*.

Parameter Declaration. The parameters of the global struct have to be defined for communication with the pipeline. We introduce the new keyword *state* to identify variables of the global struct. State parameters have to be defined with default value, to always ensure a valid definition of the struct constructor in the shader's main function. Moreover, used varying and uniform parameters have to be declared in the parameter section.

Pipeline Modification. One of the available (vertex or fragment) pipeline steps has to be specified for the placement of a custom shader function. If multiple pipeline functions subdivide a pipeline step (compare with Figure 1), the placement can be applied to a single pipeline function (substep) or to all pipeline functions (default). Four placement modes are available: *Add Before*, *Add After*, *Replace*, and *Remove*. In the *Add Before* mode, the custom pipeline function will be inserted into the selected step before the specified pipeline function and all functions, respectively. Analogously, the *Add After* mode will insert the custom function after the pipeline function. In the *Replace* mode, the custom function will replace the selected pipeline function and all functions, respectively. Finally, the *Remove* mode can be used to completely remove all functions in the pipeline step. Additionally, an optional rule can be applied to all placement modes: If the selected pipeline step is not active, the custom shader function will not be inserted into the pipeline.

String Replacement. One of our requirements is to facilitate a dynamic and flexible definition of the custom shader function. For that, a string replacement mechanism is integrated into the shader function node. The string replacement

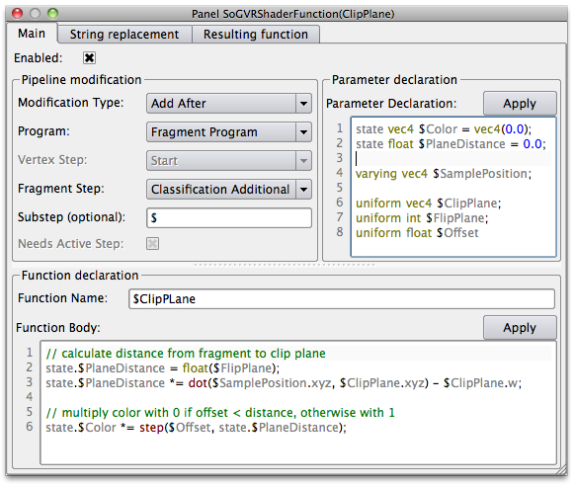


Figure 7: *The GUI of the custom shader function node. In this example, the function calculates a clip plane and modifies the color of the specified volume. Note that the character $ is a wildcard for the current volume name.*

allows replacing any string in the parameter declaration, substep, function name, and function body text fields before appending the shader function. In Figure 7, the character *$* is used as wildcard for the volume texture name. This allows for multiple instances each with a unique identifier.

5.2.2. Shader Parameter Node

If uniform parameters are used in the custom shader, the parameter values have to be bound to the OpenGL context and transferred to the graphics card. For that, shader parameter nodes with a simple GUI can be connected to the scene graph (see Figure 8). The following data types are supported: bool, 4x4 matrix, and vector[1..4] of type float and integer, respectively. Additionally, for uploading images, 1D, 2D, and 3D texture samplers as well as frame buffer sampler and cube map nodes can be used.

Figure 8: *The GUIs of three shader parameter nodes.*

5.2.3. Shader Include Node

To allow extension of the function library with custom functions or global parameters, we introduce the shader include node. With the shader include node, custom shader code can be added to the shader header of the rendering pipeline for use in custom shader functions (see Figure 9). In contrast to

the global function library, the shader include is only valid in the connected scene graph.

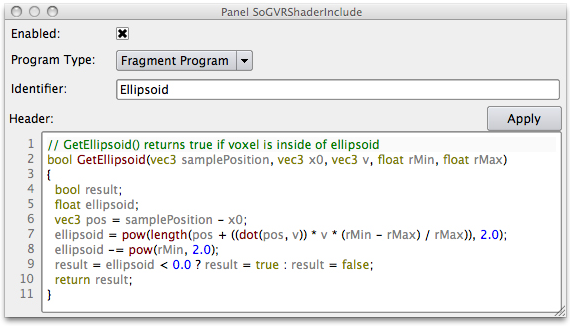


Figure 9: *The GUI of the shader include node. The function GetEllipsoid() is added into the fragment shader header.*

6. Example Applications

In this section, we present two examples for the usage of our prototyping environment. In the first example, we demonstrate how the shader pipeline can be used to implement flexible fragment clipping operations for multi-volume rendering. The second example illustrates the implementation of rendering effects for planning radiofrequency ablation.

6.1. Fragment Clipping for Multi-Volume Rendering

Fragment clipping was introduced by Weiskopf et al. [WEE02]. Visualizing multi-modal volume data typically requires the interactive definition of multiple clip planes per volume data set. Which volume is to be clipped away, and which is to be excluded from clipping must be defined for each clip plane. Designing a shader to handle this is challenging, because the exclude rule depends on the data sets currently enabled. Also, the shader should be valid if the rendering configuration changes, e.g. if a rendering effect such as shading is enabled. The following medical visualization case demonstrates our solution to integrate clip planes in multi-modal volume rendering.

A cerebral arteriovenous malformation (AVM) is an abnormal connection between veins and arteries in the brain. The complex angio-architecture of the AVM as well as the three-dimensional shape of the feeding and draining vessels has to be understood before surgery. To support the neurosurgeon in this challenging task, we developed a viewer application [WRD*11], which allows the neurosurgeon to intuitively adjust clip planes to explore the AVM, for which several data sets are considered.

Utilizing our framework, a container node has been developed to allow reuse of the clip plane in the scene graph. The container includes a single custom shader function node with the GLSL implementation of the clip plane (see Figure 7) and required parameters (e.g., plane vector and a name string of the volume to be clipped). The shader code is simple: the length from sample to plane is calculated, and the volume's color is multiplied with zero if the length is negative, otherwise it is multiplied with one. The pipeline will be modified with following setting: *Fragment Step* is set to *Classification Additional, Substep* is set to the current volume name via the wildcard *$* and the *Modification Type* is set to *Add After*. Thus, the clip plane function is always inserted after the classification of the specified volume. If the volume's classification is inactive, i.e., the volume is disabled, the clip plane function is not appended. Furthermore, a clip plane function is independent of all other pipeline function. Hence, the rendering configuration can be adjusted (e.g., varying shading modes or disabling of volumes) without invalidating the final rendering shader (see Figure 10).

6.2. Dynamic Visualization of Ablation Zones

For planning image-guided radiofrequency ablation (RFA), the cell destruction caused by ablation has to be estimated. An important requirement is that different applicator models with corresponding ablation zones, specified as ellipsoids by the manufacturers, can be visualized in the slice views and the corresponding volume rendering simultaneously. For planning of RFA, multiple applicator models with varying settings may be placed, moved, and toggled on or off in real time to discover the optimal ablation scenario.

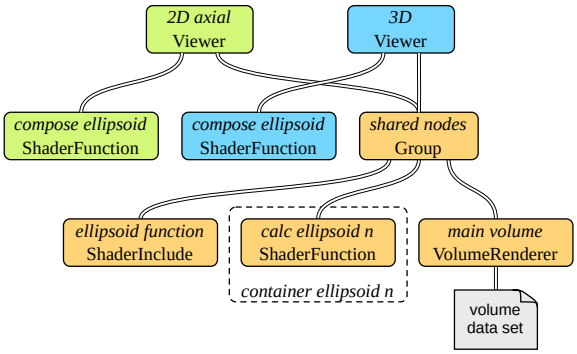
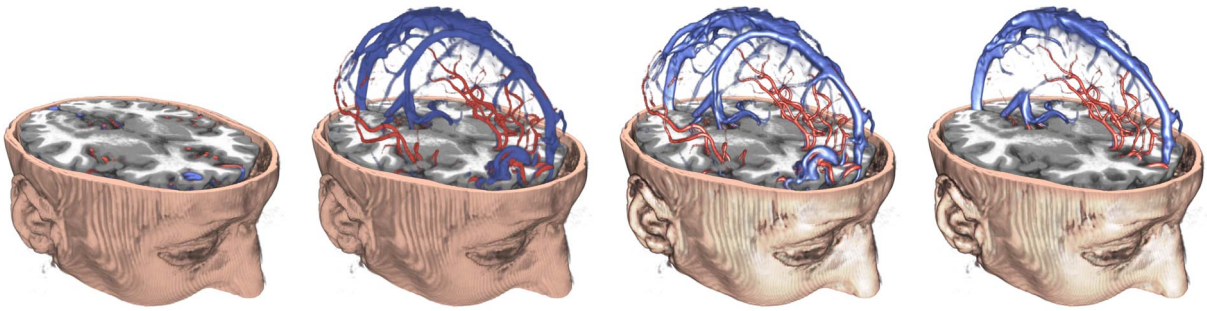


Figure 11: *An illustration of the scene graph used for the visualization of the ablation zone. Varying shader functions are used for the correct composition of the alpha value of the 2D and 3D viewers, respectively.*

For this purpose, a shader include node, which adds a custom library function into the shader pipeline, is inserted into the scene graph of our system. The custom library function evaluates whether a sample is inside or outside of the ablation zone's ellipsoid (see Figure 9). Additionally, we define an applicator container node which encapsulates a single shader function node as well as the required applicator and ablation zone shader parameters. The parameter nodes hold



(a) (b) (c) (d)

Figure 10: *Image (a) shows a multi-modal rendering of three MR data sets and a defined clip plane. The vessel data sets (red arteries, Time-of-Flight MRI; blue veins, contrast enhanced T1 MRI) are excluded from clipping (b). Image (c) shows enabled shading of all data sets. In (d), additional clip planes are defined to clip the vessels. Data sets courtesy by Lahey Clinic, Boston.*

the attributes which define the ellipsoids' size, position, and orientation. The custom pipeline function is inserted before the start step of the rendering pipeline. It calls the library ellipsoid function with the ellipsoid attributes and stores in the global state wether the current sample is inside or outside the ablation zone. To visualize the final ablation zone, we connect a single function node outside the container to handle the visual properties (color, alpha, silhouette) of the ellipsoids. The shader function is appended to the pipeline after the compositing step. This design allows altering the visualization properties for varying viewers, e.g., changing the alpha value of the ablation zone for a 2D visualization. Figure 11 illustrates the used scene graph. Note that the volume renderer and the ellipsoid header include as well as all container nodes are shared by both viewers. In contrast, the compositing of the ablation zone is implemented differently in separated shader function nodes.

We integrated this scene graph into our medical application [RSW*09]. If an applicator model is added by the end user from the application's GUI, the application automatically generates a container node with the required shader parameters and connects it to the scene graph. Thus, multiple RF applicators with corresponding ablation zones can be manipulated interactively and visualized in 2D and 3D simultaneously (see Figure 12).

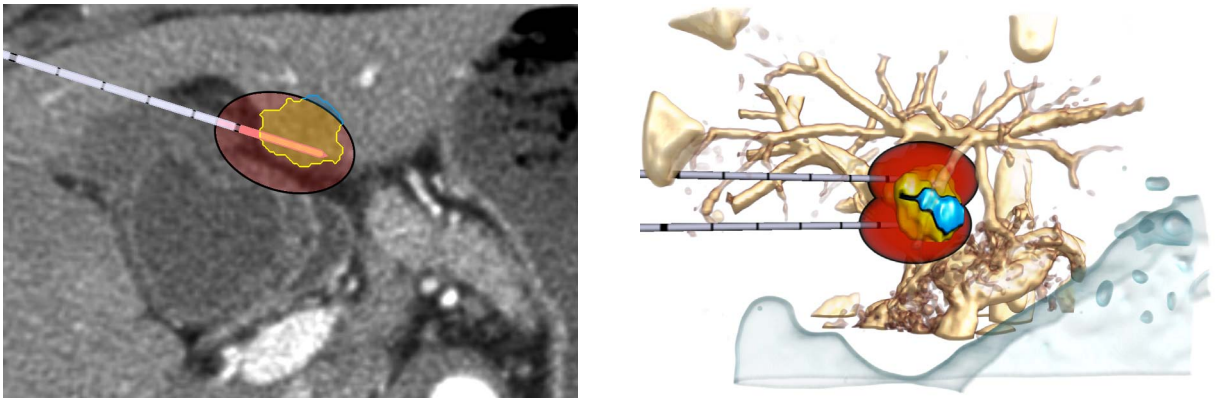
7. Results And Discussion

We present a prototyping framework for DVR, which was integrated in the *MeVisLab* development environment. In contrast to the original *SuperShader* concept, we do not use a control shader. Instead, the shader snippets are directly copied into the resulting shader string to be able to add custom shader code dynamically. To allow the compilation of the shader pipeline with custom shader code, the shader factory creates the shader string at each rendering frame and caches compiled shaders for reuse. Due to the optimization of the GLSL compiler, no changes in the final rendering performance could be measured, although the pipeline shader string is three times longer than the hand-optimized shader.

Two examples are presented, which are integrated in stand-alone medical applications developed with *MeVisLab*. In the first example, we demonstrate how a simple rendering effect can be developed and attached to the pipeline several times. In the second example, we demonstrate how varying rendering effects were developed and subsequently integrated into a medical multi-viewer software assistant. Effects such as the visualization of ellipsoids can be dynamically integrated several times in the volume rendering pipeline without considering the remaining shader. Furthermore, custom shader functions in the scene graph are used to alter the composition of the ellipsoids allowing for simultaneous visualization in different viewers (2D and 3D).

For preliminary feedback, we presented our system to a medical image processing expert, who was not involved in the development of our system. He utilized custom shader functions to implement clip planes attached to specific pipeline steps to define a dependence to the volume to be clipped. Suitable GUIs in the scene graph supported the researcher in intuitively editing custom shader code at runtime. Although the shader pipeline concept was new and the detailed internal structure of the rendering shader was unknown, the developer easily managed to implement the clip planes within a few minutes. However, basic knowledge of shader programming and volume rendering was needed.

System aim. Our framework targets supporting the developer to extend the volume rendering with custom shader algorithms and to integrate the resulting volume visualization into real-life applications with minimum development overhead. In contrast, the shader composer proposed by Plate et al. aims to support the creation of special purpose shader programs utilizing data flow widgets. The work of Rössler



(a) (b)

Figure 12: *Two RF applicators are positioned into a tumor. Image (a) shows a 2D visualization of a single slice, cutting the upper located applicator. Image (b) shows the corresponding 3D volume rendering of the same data set. Using the shader pipeline, for each representation, varying visual features are applied (colors, silhouettes, transparency), but the underlying information is equal (ellipsoid properties, image data, tumor mask). Data sets courtesy by RWTH Aachen University Hospital.*

et al. is intended to wrap complex shading algorithms into a graph representation to allow the user to concentrate on the visualization result. *VolumeShop* is a prototyping platform for visualization research that provides maximum flexibility to the developer. *Voreen* facilitates the research of new interactive visualization techniques for volumetric data sets with high flexibility. Our system is limited to customize shading effects and adjust the configuration of the render core at runtime. Extending the render core with additional, configurable features needs to be done in C++.

Modular combinability. We propose a dynamic shader pipeline, which facilitates substituting pipeline steps during run-time while ensuring a valid final shader. In other systems, the final shader is composed using predefined GLSL functions (cf. *Voreen*) or by collecting the shader snippets from predefined nodes (cf. Plate et al. and Rössler et al.). The advantage of our approach is that shader effects can be freely combined without modifying the remaining shader code or adding all possible combinations. For instance, arbitrary lights or additional volumes can be enabled, resulting in an automatic insertion of the corresponding shader code.

Extension of the rendering shader. Utilizing our shader pipeline allows the developer to interactively extend the rendering by appending arbitrary custom shaders functions without recompiling the C++ code. Utilizing the global struct for communication, the pipeline steps are independent from each other and can be changed without affecting other pipeline functions (e.g., swap gradient calculation). In contrast, the frameworks of Rössler et al. and Plate et al. do not support editing of the node's shader code without recompiling the C++ project. In *Voreen*, functions of a shader library can be interactively edited and are included in the final shader using defines. However, custom shader code has to be inserted in the final shader by hand. Similarly, *XIP* and *VolumeShop* allow editing and freely combining shader snippets. Because no mechanism for positioning of the snippets is available, the snippets also have to be manually combined to a valid shader. In conclusion, none of the related systems allows adding and editing custom shader code during runtime without the need to adjust the remaining shader (cf. arbitrary clip planes), e.g., to add a call to the shader's main function. Thus, inserting multiple custom shader code with varying parameterizations is difficult to achieve.

Application prototyping. Because our system is based on a scene graph, custom shader nodes can be dynamically connected to sub-graphs, resulting in modified rendering pipelines for different viewers (cf. compositing of ellipsoids), whereas the basic rendering shader will remain unmodified. To our knowledge, this is a unique benefit, particularly for research and development of specialized rendering effects for medical applications. Furthermore, the stability of the shader core eases the generic extension of rendering effects for long-term use in *MeVisLab*.

8. Conclusions

The presented rapid prototyping framework for GPU-based volume rendering allows flexible extension and dynamic alteration of the rendering configuration. The underlying approach is a dynamic rendering pipeline based on the SuperShader concept, to which custom shader functions can be attached at run-time. We have shown the usefulness of our framework, which has been successfully embedded into medical applications. For future work, we also plan to apply the shader pipeline to our raycasting-based volume renderer.

References

[AW90] ABRAM G. D., WHITTED T.: Building Block Shaders. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 283–288.

[BG05] BRUCKNER S., GRÖLLER M. E.: VolumeShop: An Interactive System for Direct Volume Illustration. In *Proc. of IEEE Visualization* (2005).

[BHWB07] BEYER J., HADWIGER M., WOLFSBERGER S., BUHLER K.: High-Quality Multimodal Volume Rendering for Preoperative Planning of Neurosurgical Interventions. *Visualization and Computer Graphics, IEEE Transactions on 13*, 6 (2007), 1696–1703.

[BiBPtHR08] BRECHEISEN R., I BARTROLI A. V., PLATEL B., TER HAAR ROMENY B.: Flexible GPU-based Multi-Volume Ray-Casting. *Vision, Modeling, and Visualization 2008: Proceedings, October 8-10, 2008, Konstanz, Germany* (2008), 303.

[BVUW*07] BITTER I., VAN UITERT R., WOLF I., IBANEZ L., KUHNIGK J.-M.: Comparison of Four Freely Available Frameworks for Image Processing and Visualization That Use ITK. *Visualization and Computer Graphics, IEEE Transactions on 13*, 3 (May-June 2007), 483–493.

[CJN07] CABAN J. J., JOSHI A., NAGY P.: Rapid Development of Medical Imaging Tools with Open-Source Libraries. *J Digit Imaging 20 Suppl 1* (Nov 2007), 83–93.

[Coo84] COOK R. L.: Shade Trees. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 223–231.

[ER00] EBERT D., RHEINGANS P.: Volume Illustration: Non-Photorealistic Rendering of Volume Models. In *Proc. of IEEE Visualization* (2000), pp. 195–202.

[FW04] FOLKEGÅRD N., WESSLÉN D.: Dynamic Code Generation for Realtime Shaders. In *Linköping Electronic Conference Proceedings* (2004).

[GBD04] GOETZ F., BORAU R., DOMIK G.: An XML-based Visual Shading Language for Vertex and Fragment Shaders. In *Web3D '04: Proceedings of the ninth international conference on 3D Web technology* (New York, NY, USA, 2004), ACM, pp. 87–97.

[GD06] GOETZ F., DOMIK G.: Visual shaditor: a seamless way to compose high-level shader programs. In *ACM SIGGRAPH 2006 Research posters* (New York, NY, USA, 2006), SIGGRAPH '06, ACM.

[Har04] HARGREAVES S.: Generating Shaders from HLSL Fragments. In *ShaderX^3: Advanced Rendering with DirectX and OpenGL*, Engel W., (Ed.). 2004, ch. 7.3, pp. 555–568.

[HKRs*06] HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D., ENGEL K.: *Real-Time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[LKP06] LINK F., KOENIG M., PEITGEN H.-O.: Multi-Resolution Volume Rendering with per Object Shading. In *Proc. of VMV* (2006), pp. 185–191.

[McG05] MCGUIRE M.: The SuperShader. In *ShaderX^4: Advanced Rendering Techniques*, Engel W., (Ed.). 2005, ch. 8.1, pp. 485–498.

[ME09] MCDONNEL B., ELMQVIST N.: Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *Visualization and Computer Graphics, IEEE Transactions on 15*, 6 (2009), 1105–1112.

[MeV11] MEVIS MEDICAL SOLUTIONS AG: MeVisLab, medical image processing and visualization. http://www.mevislab.de, March 2011.

[MLZ*02] MEISSNER M., LORENSEN B., ZUIDERVELD K., SIMHA V., WEGENKITTL R.: Volume rendering in medical applications: we've got pretty images, what's left to do? *Proceedings of the conference on Visualization'02* (2002), 575–578.

[MSPK06] MCGUIRE M., STATHIS G., PFISTER H., KRISHNAMURTHI S.: Abstract Shade Trees (preprint). In *Symposium on Interactive 3D Graphics and Games* (March 2006).

[MSRMH09] MEYER-SPRADOW J., ROPINSKI T., MENSMANN J., HINRICHS K. H.: Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications 29*, 6 (Nov./Dec. 2009), 6–13. to appear.

[PET07] PRIOR F. W., ERICKSON B. J., TARBOX L.: Open source software projects of the cabig^{TM} in vivo imaging workspace software special interest group. *J Digit Imaging 20*, S1 (Nov 2007), 94–100.

[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*, 6 (2007), 1584–1591.

[RBE08a] RÖSSLER F., BOTCHEN R., ERTL T.: Dynamic Shader Generation for Flexible Multi-Volume Visualization. *IEEE Pacific Visualization Symposium* (2008), 17–24.

[RBE08b] RÖSSLER F., BOTCHEN R. P., ERTL T.: Dynamic Shader Generation for GPU-based Multi-Volume Raycasting. *Computer Graphics and Applications 28*, 5 (2008), 66–77.

[RSW*09] RIEDER C., SCHWIER M., WEIHUSEN A., ZIDOWITZ S., PEITGEN H.-O.: Visualization of Risk Structures for Interactive Planning of Image Guided Radiofrequency Ablation of Liver Tumors. *Proceedings of SPIE Medical Imaging* (Jan 2009).

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. *Volume Graphics, 2005. Fourth International Workshop on* (2005), 187– 241.

[Tat04] TATARCHUK N.: RenderMonkey: an effective environment for shader prototyping and development. In *ACM SIGGRAPH 2004 Sketches* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 91–.

[TD07] TRAPP M., DÖLLNER J.: Automated Combination of Real-Time Shader Programs. In *Proccedings of Eurographics 2007* (September 2007), Cignoni P., Sochor J., (Eds.), Eurographics, The Eurographics Association, pp. 53–56.

[WEE02] WEISKOPF D., ENGEL K., ERTL T.: Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization. *Visualization, 2002. VIS 2002. IEEE* (2002), 93–100.

[WRD*11] WEILER F., RIEDER C., DAVID C. A., WALD C., HAHN H. K.: AVM-Explorer: Multi-Volume Visualization of Vascular Structures for Planning of Cerebral AVM Surgery. *Eurographics 2011. Short Papers and Medical Prize Awards* (2011), in press.

[XIP11] XIP: The Open eXtensible Imaging Platform Project. http://www.openxip.org, March 2011.