Visual Parameter Exploration in GPU Shader Space

Peter Mindek
Vienna University of Technology
Austria
mindek@cg.tuwien.ac.at

Stefan Bruckner
University of Bergen
Norway
stefan.bruckner@uib.no

Peter Rautek
King Abdullah University of Science and Technology
Saudi Arabia
peter.rautek@kaust.edu.sa

M. Eduard Gröller
Vienna University of Technology
Austria
groeller@cg.tuwien.ac.at

ABSTRACT

The wide availability of high-performance GPUs has made the use of shader programs in visualization ubiquitous. Understanding shaders is a challenging task. Frequently it is difficult to mentally reconstruct the nature and types of transformations applied to the underlying data during the visualization process. We propose a method for the visual analysis of GPU shaders, which allows the flexible exploration and investigation of algorithms, parameters, and their effects. We introduce a method for extracting feature vectors composed of several attributes of the shader, as well as a direct manipulation interface for assigning semantics to them. The user interactively classifies pixels of images which are rendered with the investigated shader. The two resulting classes, a positive class and a negative one, are employed to steer the visualization. Based on this information, we can extract a wide variety of additional attributes and visualize their relation to this classification. Our system allows an interactive exploration of shader space and we demonstrate its utility for several different applications.

Keywords

parameter space exploration, shader augmentation

1 INTRODUCTION

In data visualization, GPU shader programs are often used to process large amounts of data and to create suitable visual representations. In this case, the shaders usually implement algorithms which provide a mapping between the data and the intended visual representation. The way how the data is displayed depends on the shader program and its input parameters. The vector of values of the input parameters is a point in the corresponding parameter space. An interpretation of the resulting image requires knowledge of the relationships between the parameter space of the underlying algorithm and the visualization. These relationships might not be trivial to grasp given only the source code of the shader implementing the algorithm and the resulting image.

Data visualization algorithms implemented as GPU shaders can be investigated from various perspectives. Rendered images (i.e., image space), are presented to the user. The mapping process from data space to image space is specified in a shader program. We refer to the combination of possible shader programs and their parameters as shader space. While the image space is usually interesting for the domain expert for whom the visualization mapping was originally created, the shader space is interesting for the visualization expert. In situations where the shader program needs to be modified, it is important for the visualization expert to understand how the algorithm affects the resulting visualization. Therefore, there is a necessity for tools allowing for exploration of the shader space.

We propose a method that allows users to explore GPU shader programs and their parameter spaces by assigning semantic classifications to parts of the rendered images. This user-assigned information is subsequently used to modify the visualization mapping by generalizing it to the whole rendered image. In this way, the influence of the examined parameters or data attributes on the display of the features of interests becomes apparent.

The shader exploration is facilitated by a well defined domain specific language (DSL) that extends the shader language under consideration. The DSL is used to annotate the shader program without changing its functionality. The annotations are inserted as comments that are parsed and interpreted by our system. This strategy is similar to well known documentation tools like Doxygen and JavaDoc. The benefits are:

- the annotations are kept close to the original code (resulting in easier maintainability)

- the annotations are transparent to the host language.

We refer to the insertion of comments that include commands from the DSL as shader augmentation. The augmented shader can either be run unmodified to fulfil its original purpose, or be transformed for the shader space exploration. This provides functionality for the exploration of desired parameters and data attributes. As the additional functionality is part of the augmented shader, it is executed on the GPU, which benefits the performance of the system.

The main contributions of this paper are:

- a method for translating semantic classifications from image space to shader space and using them to explore the effects of parameters on the rendered images
- a parameter exploration language (PEL), which is used to extract values of relevant parameters and data attributes from the shader program for further parameter-space exploration
- a graphical user interface for the automatic and transparent generation of PEL code.

While the PEL can be employed by visualization experts, the graphical user interface allows the shader-space exploration even without knowledge of the shading language, which might be a benefit for domain experts.

2 RELATED WORK

Our method is designed for exploring parameter spaces of various visualization algorithms. The method is particularly applicable to volume rendering. Volume rendering algorithms are usually complex and it is difficult to predict the effects of their parameters and data attributes. In this work, we focus mainly on volume rendering as a possible application area for our method.

In volume rendering, the mapping between data and rendered images is usually adjusted using transfer functions. A transfer function maps one or more data attributes to optical properties. It enables users to visually examine these attributes of the data. Wu and Qu [20] present a framework for interactive transfer function design based on genetic algorithms and image similarity, where the user edits direct volume rendered images. Another approach is presented by de Moura Pinto and Freitas [3]. It is based on dimensional reduction of the voxel attributes using self-organizing maps. Correa and Ma [1] propose visibility-driven transfer functions for enhancing the visibility of important features in volume data. Tzeng and Ma [19] propose an interface for data classification in a cluster space of different materials present in the data.

Tzeng et al. [18] propose a volume data classification approach based on machine learning. This approach employs a sketch-based interface to select a primary classification, which is then used as a training set for a machine learning algorithm. Similar work is also presented by Guo et al. [7]. Their WYSIWYG (What You See Is What You Get) approach allows definition of one-dimensional transfer functions by direct interaction with the rendered images. Yuan et al. [21] present a method for sketch-based volume segmentation. Visualization mapping based on semantics is proposed by Rautek et al. [14]. In this method, the mapping of the data attributes to visual styles are described by domain experts in the natural language. An extension of this work [15] allows to use interaction-dependent rules for specifying the visualization mapping.

Gavrilescu et al. [5] present a work on user interfaces offering information on the effects of parameters that are adjusted by these interfaces. Our method also enables to explore the effects of the parameters. However, we provide a possibility to observe the effects on specified features in the visualization. This makes our method useful not only in data exploration, but also in the exploration of the visualization algorithms and their behaviour.

The visualization-algorithm analysis capabilities of our method can be used for debugging purposes as well. Various systems for analyzing and debugging visualization software have been presented [4] [8] [16]. Crossno and Angel [2] present a case study on debugging of visualization software. Meyer-Spradow et al. [13] propose a framework for rapid prototyping of visualization algorithms by connecting modules into a data-flow network. The framework also provides debugging abilities by displaying outputs of the individual modules. Our method allows to visually identify effects of particular shader parameters and data attributes on the rendered images. Rather than displaying values of individual variables, it provides visual feedback on how the values correspond to the user-defined classification of the rendered image. These findings can be compared to expected behaviours for debugging purposes or the visualization-algorithm analysis. Our method can be used as a complementary tool to existing shader debugging solutions.

Our work is related to data-exploration methods, since it uses similar principles to explore the shader space of the visualization algorithms. Jankun-Kelly et al. [10] propose a model for the visualization exploration process. McCormick et al. [11] propose Scout - a visualization system utilizing the GPU for exploration and visualization by applying queries directly to the data. Gerl et al. [6] propose a method where properties of visualization mappings are rendered, and brushing is used on these rendered images to explicitly spec-

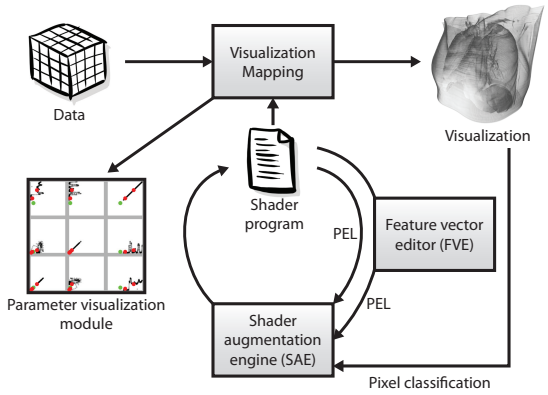


Figure 1: Overview of the visual shader-space exploration.

ify semantics for volume visualization. McDonnel and Elmqvist [12] present the concept of using the GPU for information visualization. They propose an interface for creating visualizations on the GPU without shader-programming knowledge. Jankun-Kelly and Ma [9] propose an interface for parameter space exploration using a spreadsheet-like visualization. The authors show several renderings with different parameter settings which can be used to explore the parameter space. We apply our method to investigate the effects of parameters of particular visualization algorithms on the resulting image. Knowledge gained from this process can be used in the exploration of other datasets using the visualization algorithm.

The goal of our method is to allow comprehensible analysis of visualization algorithms. Our approach is conceptionally similar to some of the mentioned techniques which allows to specify the mapping from data attributes to visual representation in a flexible manner. In contrast, our method allows to modify an existing visualization mapping by generalizing user-defined positive and negative examples in the rendered image, while influence of examined parameters and data attributes is taken into account. This possibility makes our method capable of complex exploration of the shader space, which is difficult to carry out using the existing methods for explicit specification of the visualization mapping.

3 VISUAL SHADER-SPACE EXPLORATION

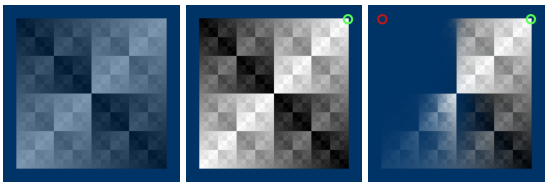
The usual way to explore visualization algorithms is through examination of the results, i.e., rendered images. The effects of various isolated parts of the visualization algorithm are difficult to comprehend by looking at the image space only. However, if the rendered image reveals features of the visualized data, the user can identify areas of the image where the features are visible and areas where they are occluded or not displayed correctly. These areas are positive and negative examples of the visualization-algorithm behaviour. For the purpose of visualization-algorithm exploration, it is interesting to identify variables of the algorithm which could be used for generalizing the behaviour from the positive examples for the whole rendered image. We propose a method for fast determination of whether a particular set of variables can generalize behaviour of the algorithm in a given positive example area for the whole image.

In our visualization-algorithm exploration-method the user interaction is limited to the identification of positive and negative examples in the rendered image, and isolating one or more parameters or data attributes which should be used to generalize those examples. The goal is to help emphasize relationships between various parameters of the visualization algorithms and data features displayed by them. The purpose of the method is to help analyze visualization mappings provided by GPU shader programs and to find ways how the mappings could be enhanced. We call our method visual shader-space exploration (VSSE).

Figure 1 shows the overview of VSSE. An image is created by transforming the data through the visualization mapping. The visualization mapping is implemented by a shader program. Subsequently, the visualization mapping can be modified by the user. In order to modify the visualization mapping, the parameters of the original shader which are to be examined are selected. This can be done either using the Parameter Exploration Language (PEL) which is embedded into the original shader code, or using the Feature Vector Editor (FVE). The FVE also generates PEL code, so the PEL and the FVE can be used simultaneously. The PEL code in the original shader is then parsed with the Shader Augmentation Engine (SAE) and an augmented version of the shader is generated.

The augmented version of the shader provides the original visualization as well as the possibility to select several pixels in the rendered image (i.e., image space of the visualization) where features of interest are visible, and several pixels where they are occluded. We refer to the pixels displaying the features of interest as the positive class, while the other ones constitute the negative class. Subsequently, the way how selected parameters and data attributes are used in the visualization mapping is modified so that the selected examples are generalized to the whole image. This approach constitutes simple yet effective means for analysing the influence of the parameters and the data attributes to features observed in the rendered image.

For every classified pixel a feature vector based on the values of the examined parameters is extracted. The classification of the pixel is then assigned to the extracted feature vector. VSSE provides means to calculate a fuzzy classification, or a membership degree of a



(a) (b) (c)

Figure 2: An example demonstrating the usage of VSSE with a simple shader. In (a), no pixels are classified, opacity is 0.5. In (b), a positive pixel is selected (green circle), opacity is 1 for the whole image. In (c) additionally a negative pixel (red circle) is selected.

feature vector to either the positive or the negative class in any state of the execution of the shader. The membership degree is in the interval $[0, 1]$, where 0 means association with the negative class and 1 means association with the positive class. The value of the membership degree can be freely used in the shader to modify the visualization mapping so that the classification of the rendered pixels is displayed. We also refer to the membership degree as *confidence*.

Figure 2 demonstrates our method on a simple shader displaying a texture. In this case, the feature vectors are composed of texel luminance and coordinates. The confidence is used to modulate the opacity of the texels. In Figure 2(c), two corners of the texture have been classified, one as a positive example, another one as a negative example. The result is a clear separation of the black (top left) from the white (top right) squares. The white square is fully opaque, while the black square is completely transparent. In this example, VSSE shows how texel luminance and its coordinates affects various parts of the rendered texture and how the parts can be separated by setting up one positive and one negative example.

4 PARAMETER EXPLORATION IN SHADER SPACE

In VSSE, the shader-space exploration is carried out by visualizing how parts of the examined shader influence displayed data features. Every pixel of the final image is calculated by a single instance of the visualization shader. The instance is described by a feature vector. The feature vector consists of values extracted during execution of the particular shader instance. Any expression of the shading language can be used for these values. In this way it is possible to extract values of variables or functions at particular positions within the shader program, even in loops or conditional statements. It is also possible to extract multiple feature vectors for a single instance. In that case, every feature vector describes a particular state of the shader program. For instance, in volume rendering, the color of every pixel is determined by compositing several data samples along a ray. For each data sample, a feature vector describing the current sample can be extracted.

The user can classify some of the extracted feature vectors as positive or negative examples. Our method enables the generalization of this classification to all feature vectors extracted during the rendering of the final image. We propose to employ this generalization to modify the visualization mapping. This way, the influence of the selected parameters, data attributes, and intermediate results of the shader on the rendered image can be explored without rewriting the shader. A benefit of our method is also the possibility of taking values of previous executions of the shader into account. Without using our method, this would have to be implemented manually by writing values to the GPU memory and reading them back.

4.1 PEL - Parameter exploration language

We propose a parameter exploration language (PEL), which is the language of shader annotations used for the parameter exploration. The purpose of the PEL is to mark which parameters or data attributes used in the shader program should be examined and how should their effects on the visualized data be displayed. The advantage of using PEL over simply rewriting the shader program in the desired way is that the PEL allows to store multiple values of the examined parameters and data attributes during the shader execution. These values can be stored, classified as positive or negative examples, and used in subsequent executions of the shader program. By using the PEL, this functionality is added to the shader program automatically and there is no need in explicitly creating it.

The PEL can be embedded into the shader source code as shading language comments. Both line and block comments are supported. Comments starting with a dot are intepreted as PEL annotations. They are transformed to regular shader code by the shader augmentation engine. This way, expressions of the PEL and the shading language can be easily combined.

A feature vector can be extracted using the following annotation:

```
//. vector p[0]; ... p[n]; weight
```

This annotation is replaced by code that evaluates the expressions *p[0]*, ... *p[n]* and *weight* and extracts them as the corresponding feature vector.

To allow a flexible specification of how to visualize effects of the examined parameters, the keyword *confidence* can be used in PEL annotations. This keyword

is replaced with a function that evaluates the membership degree for the most recently extracted feature vector. The value represents likeliness of the current shader program state to belong to the positive or the negative class. This value can be arbitrarily displayed by the shader program.

The following listing shows a part of the fragment shader source for the example in Figure 2:

```
vec4 color = texture2D(tex, co.st);
//.vector color.r; co.s; co.t; 1.0
fragColor = vec4(vec3(color.r),
            1.0 /*. - confidence */);
```

In the listing the texel luminance (*color.r*) and coordinates (*co.s, co.t*) define a feature vector. In this example, one feature vector is extracted for every rendered pixel. Some of the pixels are user-classified as negative or as positive. For every other pixel, the confidence of the respective feature vector is calculated and used as opacity (*confidence* keyword) for the pixel.

A feature vector can be extracted in every stage of the shader program. During the execution of the shader, the feature vector may be extracted multiple times for a single pixel. The pixel can be classified as a positive or a negative example by the user. Since the color of the pixel depends on all of the extracted feature vectors, they have to be composited into one feature vector representing the pixel. We refer to it as pixel feature vector. The pixel feature vector is assigned the classification that was specified for the pixel by the user. Various user-selectable composition algorithms, or accumulation types, can be applied. These are defined using *accumulation* keyword of the PEL. The default accumulation type is weighted average using weights assigned to individual extracted feature vectors.

The composition of feature vectors is helpful for applications where the color of the pixel is influenced by multiple data samples, for example volume rendering. The accumulation type should reflect the strategy used for calculating the color of the pixel. Assigning a classification to the pixel feature vector ensures that the positive or negative example is suitably placed in the feature vector space.

The augmented version of the shader can be switched to one of two modes. The first mode is the rendering mode without processing of user input. In this mode, feature vectors are extracted, and the membership function is evaluated for the most recently extracted feature vector. The extracted feature vectors are not composited into pixel feature vectors and no classification is assigned to them. The second mode is the classification mode. In this mode, one pixel and its user-defined classification has to be selected. For the pixel that is being classified, extracted feature vectors are composited into

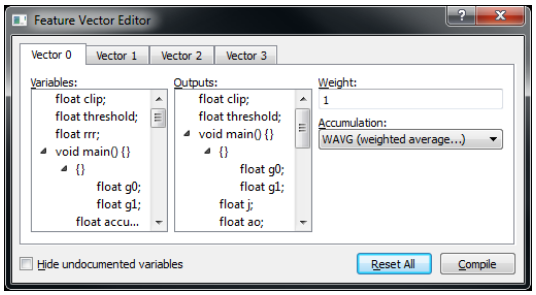


Figure 3: Feature vector editor (FVE).

a classified feature vector at the end of the execution of the shader. The purpose of this design is to enable the host application to use the same shader for rendering as well as user-input processing (classification of feature vectors).

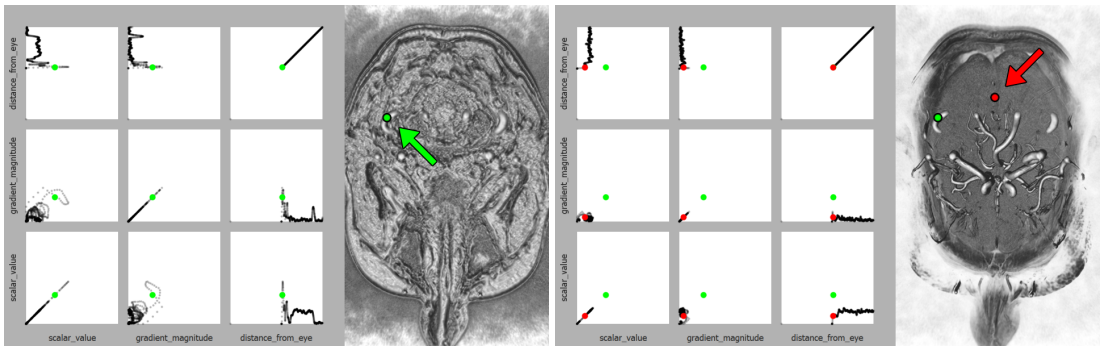
4.2 FVE - Feature vector editor

The PEL offers a high degree of flexibility when designing a parameter exploration scenario for a particular shader program. However, it requires users to actually understand the shader source code in order to create meaningful PEL annotations. To address this issue, we propose a Feature vector editor (FVE). It is a graphical user interface which allows users to create and extract feature vectors from the shader without having to understand it. The FVE interface is shown in Figure 3.

The FVE parses the shader source code and extracts all floating point variables. The extracted variables are then presented to the users so that they can pick any of them to form a feature vector. In the shader code, the formed feature vector is extracted after the last of the selected variables is defined. This means that the variables used for the feature vectors should be initialized with a meaningful value. This is one of the limitations of the FVE. It is not as flexible as using PEL directly, where the user can extract feature vectors from any place in the shader code. Another limitation is that the user can take only variables for the feature vectors, while in PEL any language expression is possible.

The FVE displays the names of the variables as extracted from the shader source code. These names may be confusing for the user and documentation is needed for the effective usage of the FVE. Therefore, we have included two special annotations, *name* and *desc* in the PEL. These annotations are provided for programmers to document individual variables directly in the shader source code so that it is convenient to explore the shader program using FVE. These annotations can change a displayed name of a particular variable and add a description to it. The following listing shows an example:

```
//. name Brightness
//. desc Brightness of the pixel
float br;
```



(a) (b)

Figure 4: A scatterplot matrix displaying feature vectors. Positive vectors are marked with green circles, negative ones with red circles. (a) shows a magnetic resonance angiography dataset of a human brain, where a pixel displaying a vessel was classified as positive (green circle). In (b), a pixel inside the brain matter is classified as negative (red circle), which results in an enhanced view of the vessels. VSSE confirmed that the explored parameters can effectively classify blood vessels in the brain.

The two PEL annotations in the first two lines cause FVE to display *Brightness* instead of *float br;* in the list of variables. Additionally, the FVE displays a pop-up text with the given description on hovering the mouse over the variable.

This mechanism creates a level of abstraction between the shader program source and the parameter exploration. It is the responsibility of the programmer of the shader to provide names and descriptions of variables so that a domain expert can understand them. By using reasonable names and descriptions, the domain expert can explore the shader space without shading-language knowledge, or without a deep understanding of the particular shader program.

We designed our method in such a way that it can benefit domain experts and visualization experts. Visualization experts can use PEL or FVE for analysing their shaders by examining effects of selected parameters. The FVE can be also employed for fast prototyping of new algorithms or visualization mappings derived from the original shader without recompiling the host application.

The visualization expert can prepare the shader for the exploration carried out by the domain expert by adding reasonable domain-specific variable names and descriptions using PEL annotations without actually renaming the variables. The domain expert can then use FVE for exploring the visualization mapping by changing relationships between its parameters and data attributes. The shader program is transparent to the domain expert because the FVE shows the domain-specific names and descriptions of individual variables available for the exploration.

The flexibility of the FVE is not as high as the one provided by PEL directly. In case the FVE is not sufficient in a certain scenario, it is possible to use the PEL annotations together with the FVE. Naturally, in this case knowledge of the shader program is needed.

4.3 Visualization of parameter space

During classification, many potentially multi-dimensional feature vectors are stored in the GPU memory. In addition to visualizing their effects on the rendered image, it is also useful to examine their relationships. For instance, when VSSE is used for identifying suitable parameters for a multi-dimensional transfer function, the parameters of the examined shader program are sequentially chosen for the feature vectors. When a good set of parameters is identified, it is necessary to check whether some of the parameters are correlated. If a pair of the parameters correlates, one of them can be omitted from the designed transfer function.

We introduce a visualization module which is able to depict all positively and negatively classified feature vectors, as well as extracted feature vectors for the currently selected pixels as a context. The visualization module uses a scatterplot matrix, which provides an overview of the bilateral relationships between elements of the vectors.

The visualization module is shown in Figure 4. In this case, feature vectors are composed of following values: scalar data value (*scalar_value*), gradient magnitude (*gradient_magnitude*), distance from the virtual camera (*distance_from_eye*). The goal is to determine whether these three data attributes can be used for effective classification of blood vessels in an MRA scan of a brain. A part of a displayed blood vessel is classified as a positive example for being a vessel. Part of brain matter is classified as a negative example for being a vessel. The opacity of all rendered voxels now depends on how close their attributes are to the specified positive and negative examples.

The user of our method selects examples for the positive and the negative class manually. It is possible to classify several different materials as a positive or negative class. In this case, our method can reveal if the selected variables of the feature vector can provide unified description of these materials.

The visualization module shows the relation of each pair of variables of the feature vector in a scatterplot matrix. For every user-defined classification, multiple values of the variables are extracted and composited into a feature vector. The visualization module can be used to determine the best composition strategy for calculating the feature vectors. In the example of Figure 4, the scatterplots with the *distance to the virtual camera* on one of the axes show peaks on the positively classified pixel, while there are no peaks in the negatively classified one. As the axis of the scatterplots is the distance to the virtual camera, the scatterplots actually show ray profiles. The ray profiles of both the positive and the negative example begin with values close to zero. This means the feature vectors should be composited using a weighted average, otherwise the positive and the negative examples could not be distinguished.

5 FEATURE VECTORS

The user can classify multiple pixels in one session. Therefore, the shader has access to multiple feature vectors associated with either class. At any point of the execution of the shader, this information can be used to calculate the confidence that a most recently extracted feature vector belongs to the positive or the negative class. The confidence is determined by a membership function.

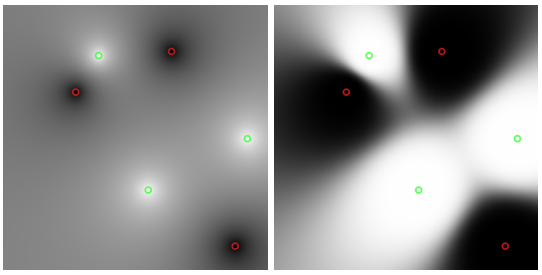
The membership function is used for an automatic classification of pixels that have not been classified by the user as either positive or negative. The function should be smooth and it should not be sensitive to slight changes in its inputs, since these are provided only with a certain accuracy through interaction in image space. For this purpose, Euclidean distances of the classified feature vectors to the pixel feature vector can be employed.

The membership function is defined for pixel feature vectors $\bar{x}_i$ as $f(\bar{x}_i) = c_i$ where c_i is the user defined classification for the feature vector of pixel i. The classification is given as follows:

$$c_i = \begin{cases} 1 & \text{if positive} \\ 0 & \text{if negative} \end{cases} \quad (1)$$

For feature vectors not classified by the user ($\bar{x} \neq \bar{x}_i$), the membership function is defined as follows:

$$f(\bar{x}) = \frac{\sum_{i=1}^{n} (c_i \frac{1}{\|\bar{x}-\bar{x}_i\|^p})}{\sum_{i=1}^{n} (\frac{1}{\|\bar{x}-\bar{x}_i\|^p})} \quad (2)$$



(a) (b)

Figure 5: Visualization of a two-dimensional case of a membership function for (a) $p = 1$ and (b) $p = 5$. Black color means association with the negative class, white color means association with the positive class. Green circles denote positive feature vectors, red circles denote negative ones.

where $\bar{x}_{1..n}$ are the user-classified feature vectors. p is an additional parameter that adjusts softness of the membership function. This fine-tuning is useful when the VSSE is applied to different types of data. Figure 5 gives an example of a two-dimensional feature-vector membership-function with $p = 1$ and $p = 5$.

If there are no user-classified feature vectors yet, the confidence for any feature vector cannot be evaluated. In this case, the membership function is defined as $f(\bar{x}) = 0.5$.

6 IMPLEMENTATION

The system is implemented in C++ using the Qt library. It provides an application programming interface for an easy integration into existing host visualization systems.

The shader augmentation engine is able to enhance shaders written in the GLSL language by replacing PEL annotations with GLSL code. The inserted GLSL code uses the EXT_shader_image_load_store extension for storing and loading feature vectors in OpenGL textures.

When there is a request from the host application to classify a pixel, the host application has to provide the augmented shader with the screen space coordinates and the desired class (positive or negative) of the selected pixel. The augmented shader is then automatically switched to the classification mode. The pixel has to be rendered in this mode, so that its pixel feature vector is calculated, classified, and stored. Other pixels can be rendered in the classification mode as well, but the code for feature vector extraction would be ignored for any pixel with different screen space coordinates from the pixel that was selected (e.g., by mouse clicking on the rendered image).

The parameter visualization module is implemented using GLSL shaders to access extracted feature vectors and render them in a scatterplot matrix. A geometry

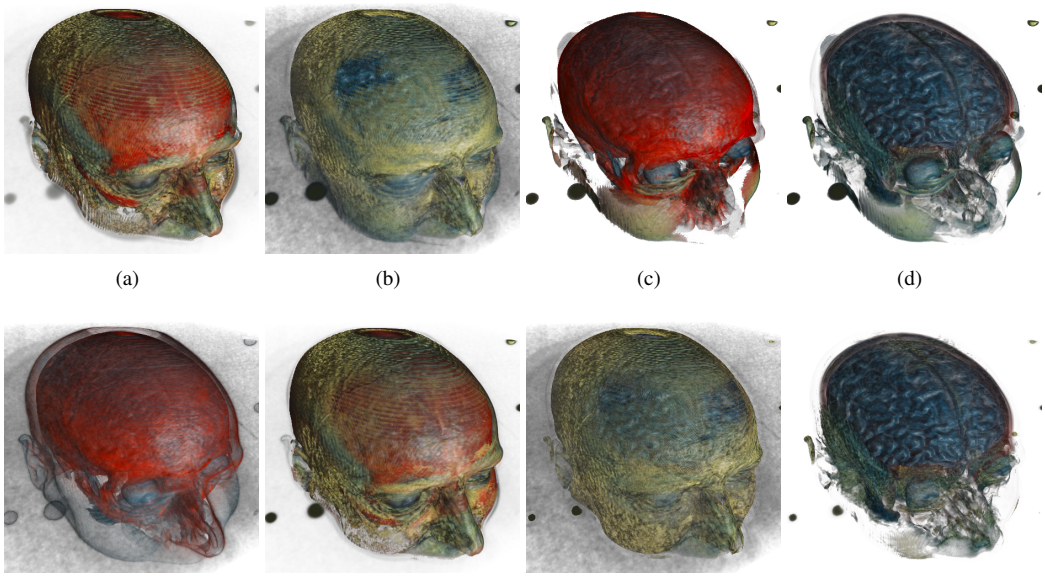


Figure 6: Visualizations of the dual-modality data with positively classified brain and negatively classified occluding tissues for different feature vectors. Feature vectors consist of these attributes: (a) voxCT; (b) gmCT; (c) voxDiff; (d) gmCT, voxDiff; (e) voxMRI; (f) gmDiff, voxMax; (g) gmDiff, gmMax; (h) gmMax, voxDiff

shader is used to create the visualization from the data stored in the GPU memory to minimize data transfer between GPU and CPU.

7 USE CASES

In the following section we present two use cases of VSSE. We apply our method to two distinct shader programs to demonstrate its versatility. In both examples, our method is used to examine an existing algorithm with different goals.

7.1 Volume Rendering

To demonstrate the shader-space exploration-abilities of our method, we employ it for the exploration of a direct volume rendering algorithm. For this use case, we use dual modality data - a co-registered CT and MRI scan of a human head. The MRI data is displayed on the screen, while the CT data is used only to extract additional data attributes. We use VSSE to find out which of the data attributes can be employed to design a transfer function for the effective classification of the brain in this type of data.

We identified several data attributes to be taken under consideration. The data attributes are: scalar value from the MRI data (*voxMRI*), scalar value from the CT data (*voxCT*), gradient magnitude from the MRI data (*gmMRI*), gradient magnitude from the CT data (*gmCT*), difference of the two scalar values (*voxDiff*), difference of the two gradient magnitudes (*gmDiff*), maximum of the two scalar values (*voxMax*), maximum of the two gradient magnitudes (*gmMax*).

Using VSSE, we extract feature vectors for every processed voxel. The feature vectors are formed from various combinations of these data attributes. The method for the composition of the feature vectors is weighted average. The weight for every feature vector is the contribution of the respective voxel to the final pixel color. Therefore, feature vectors composited using this strategy accurately describe the pixels. Finally, we use the confidence value to modulate the opacity of every processed data sample. This means that data samples with feature vectors from the negative class do not contribute to the pixels of the rendered image, thus they do not occlude areas of interest.

Using the FVE we select a set of variables representing the data attributes to form a feature vector. We try various different feature vector setups in order to find most suitable ones. Using a clipping plane, we reveal brain in the visualization of the MRI data. We positively classify one pixel inside the brain. Additionally, three pixels in the skull and other occluding tissues are classified negatively. We use the same user-specified classification for every feature vector setup. Since we classified a pixel displaying the brain as positive, and pixels displaying occluding tissues as negative, the resulting visualization should reveal the brain. If this is not the case, we can conclude that the current set of data attributes would not be suitable for designing a transfer function for the desired visualization mapping. By trying different sets of data attributes, we identify those suitable for classifying brain. Figure 6 shows the results. The best choices are shown in Figure 6(d) and Figure 6(h), where the brain is revealed as expected.

7.2 Image processing

To demonstrate the generality of our method, we show how it can be used to analyse an extension of an image processing algorithm. The goal is to determine whether the extended algorithm is capable of generalising behaviour specified by positive and negative examples.

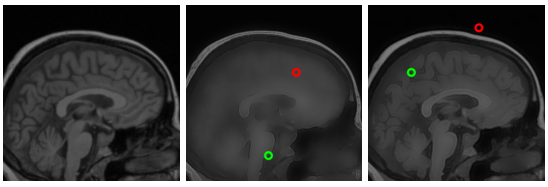
In this example, a shader implementing a bilateral filter [17] is used. The bilateral filter smooths images while preserving edges. It replaces every pixel of the input image with weighted average of surrounding pixels. The weights are determined by a two-dimensional Gaussian function and a color difference between the original and the surrounding pixel. If the difference is higher than a threshold, the weight of the pixel is zero.

The bilateral filter has two parameters: blurring radius and threshold. Higher radius will result in removal of more high-frequency noise, while the threshold determines how strongly should the edges be preserved.

We use our method to explore possibilities of extending the bilateral filter algorithm by using different threshold for every pixel of the rendered image. The goal is to see how the extended algorithm could emphasize features or areas of interest of the images from different application domains. Our method is used to specify these features by classifying several pixels of the image as the positive and the negative examples. Afterwards, it is observed how the classification has been generalized for the rest of the image to evaluate the usefulness of the extended algorithm.

Since the threshold controls how edges are preserved, we apply the Sobel operator to calculate gradient magnitude for every pixel and use it as a feature vector. The gradient magnitude is high for edges and low for flat areas of the image. In this setup, the user can select examples of edges (positive examples) and flat areas (negative examples) by clicking on them. These examples are subsequently generalized for calculating the threshold for each pixel.

Figure 7 shows two different selections of edge examples as well as the original image. In 7(b) only strong edges are visible. In 7(c), a more subtle edge was selected as a positive example. The thresholds were modified so that more edges are visible. Our method shows the potential of the proposed extension of the bilateral filter algorithm for preserving only the edges with a specific significance.



(a) (b) (c)

Figure 7: Using VSSE for exploring possibilities of variable threshold of a bilateral filter. In (a) the original image is shown. In (b) and (c) one positive example of an edge was chosen (green circle) and one negative (red circle). The thresholds for individual pixels are modified so that the examples are generalized for the whole image.

8 DISCUSSION AND LIMITATIONS

Our proposed method is intended for analyzing existing visualization algorithms in order to gain better insight into their inner working. As we show in section 7.1, our method enables rapid exploration of shader space of an visualization algorithm resulting in identification of data-attributes combinations interesting for a specific field. Achieving this goal by simple shader programming would have to be carried out by cumbersome manual evaluation of effectiveness of transfer functions for each visualization mapping. In section 7.2 we show that our method can be employed for a different type of shader analysis as well.

There are several limitations in our method that should be addressed in future work. Currently, the shader augmentation engine is able to parse only GLSL shaders. However, the presented concepts do not depend on using shaders. There is no principal obstacle to provide implementations for other languages implementing the visual mapping.

We have evaluated the performance of the system on a volume rendering shader displaying various datasets. For a dataset of 424x279x190 voxels and a window size of 800x600 pixels, the shader was running at around 120 FPS. After augmenting the shader with the PEL annotations and classifying five pixels using three-dimensional feature vectors, the FPS dropped to approximately 20 FPS. This performance drop is caused by additional GPU memory accesses for extracting values for the feature vectors and reading them back to the CPU for the calculation of the classification of every processed voxel. The timings were obtained with a GeForce GTX 480 graphics card.

9 CONCLUSION

We have proposed visual shader-space exploration, a method for the visual analysis of GPU shader programs. The method enables users to explore effects of various parameters of visualization algorithms. It provides a visual feedback based on user-specified semantic classification of parts of the rendered images. We have implemented the method as a set of C++ classes that are independent of the underlying visualization system.

The implementation could be further enhanced by adding support for different language back-ends (such as OpenCL or CUDA). Various optimizations would

also be possible to improve performance on large datasets, e.g., using caching for the membership functions.

10 ACKNOWLEDGMENTS

The work presented in this paper has been partially supported by the ViMaL project (FWF - Austrian Research Fund, no. P21695) and by the Aktion OE/CZ grant number 64p11.

11 REFERENCES

[1] C. D. Correa and K.-L. Ma. Visibility histograms and visibility-driven transfer functions. *IEEE Transactions on Visualization and Computer Graphics*, 17:192–204, 2011.

[2] P. Crossno and E. Angel. Visual debugging of visualization software: a case study for particle systems. In *Proceedings of the conference on Visualization '99*, VIS '99, pages 417–420. IEEE Computer Society, 1999.

[3] F. de Moura Pinto and C. M. D. S. Freitas. Design of multi-dimensional transfer functions using dimensional reduction. In *Eurographics - IEEE VGTC Symposium on Visualization*, pages 131–138, 2007.

[4] N. Duca, K. Niski, J. Bilodeau, M. Bolitho, Y. Chen, and J. Cohen. A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics*, 24(3):453, 2005.

[5] M. Gavrilescu, M. M. Malik, and M. E. Gröller. Custom interface elements for improved parameter control in volume rendering. In *14th Int. Conf. on System Theory and Control 2010*, pages 219–224, 2010.

[6] M. Gerl, P. Rautek, T. Isenberg, and E. Gröller. Semantics by analogy for illustrative volume visualization. *Computers & Graphics*, 36(3):201–213, 2012.

[7] H. Guo, N. Mao, and X. Yuan. Wysiwyg (what you see is what you get) volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2106 –2114, 2011.

[8] Q. Hou, K. Zhou, and B. Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. *ACM Trans. Graph.*, 28(5), 2009.

[9] T. J. Jankun-Kelly and K.-L. Ma. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, 2001.

[10] T. J. Jankun-Kelly, K. L. Ma, and M. Gertz. A model for the visualization exploration process. In *Proceedings of the conference on Visualization '02*, VIS '02, pages 323–330. IEEE Computer Society, 2002.

[11] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 171–178. IEEE Computer Society, 2004.

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

[13] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs. Interactive design and debugging of gpu-based volume visualizations. In *Computer Graphics Theory and Applications*, pages 239–245, 2010. short paper.

[14] P. Rautek, S. Bruckner, and E. Gröller. Semantic layers for illustrative volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1336–1343, 2007.

[15] P. Rautek, S. Bruckner, and M. E. Gröller. Interaction-dependent semantics for illustrative volume rendering. *Computer Graphics Forum*, 27(3):847–854, 2008.

[16] M. Strengert, T. Klein, and T. Ertl. A hardware-aware debugger for the opengl shading language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 81–88. Eurographics Association, 2007.

[17] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. *Sixth International Conference on Computer Vision*, pages 839–846, 1998.

[18] F.-Y. Tzeng, E. Lum, and K.-L. Ma. An intelligent system approach to higher-dimensional classification of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):273–284, 2005.

[19] F.-Y. Tzeng and K.-L. Ma. A cluster-space visual interface for arbitrary dimensional classification of volume data. In *Eurographics - IEEE TCVG Symposium on Visualization, 2004*, pages 17–24, 2004.

[20] Y. Wu and H. Qu. Interactive transfer function design based on editing direct volume rendered images. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1027–1040, 2007.

[21] X. Yuan, N. Zhang, M. X. Nguyen, and B. Chen. Volume cutout. *The Visual Computer (Special Issue of Pacific Graphics 2005)*, 21(8–10):745–754, 2005.