Institut für Computergraphik und Algorithmen

Technische Universität Wien

Karlsplatz 13/186/2
A-1040 Wien
AUSTRIA

Tel: +43 (1) 58801-18601
Fax: +43 (1) 58801-18698

Institute of Computer Graphics and Algorithms

Vienna University of Technology

*email*:
technical-report@cg.tuwien.ac.at

*other services*:
http://www.cg.tuwien.ac.at/
ftp://ftp.cg.tuwien.ac.at/

# TECHNICAL REPORT

## Interactive Algorithm Exploration using Meta Visualization

P. Sikachev[1], A. Amirkhanov[1], R. S. Laramee[2], and G. Mistelbauer[1]
[1]Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria
[2]Department of Computer Science, Swansea University, Swansea, Wales, United Kingdom

TR-186-2-11-2

October 2011

# Interactive Algorithm Exploration using Meta Visualization

P. Sikachev[1], A. Amirkhanov[1], R. S. Laramee[2], and G. Mistelbauer[1]
[1]Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria
[2]Department of Computer Science, Swansea University, Swansea, Wales, United Kingdom

October 20, 2011

## Abstract

In volume rendering shadows provide an important visual cue, and enhance depth perception. However, shadowing has its own disadvantages. Shadows do not take into account the importance of features and could potentially result in too dark scenes.

In this paper we propose an approach for inconsistent shadowing, which is designed to overcome these limitations. For the purpose of illustrative rendering, we propose two complementary techniques: 1. shadow caster inconsistency and 2. shadow receiver inconsistency. We demonstrate several advantages of the different approaches, using inconsistent shadowing. We present two approaches, based on shadow transfer function concept, an approach, introducing usage of gradient magnitude information in shadowing, and a method for adaptive shading and shadowing of the surface, depending on the gradient certainty.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and RealismColor, shading, shadowing, and texture

## 1 Introduction

In recent years, rendering and visualization algorithms, or *display algorithms*, have become increasingly complex. The growth of GPU's performance together with new shader language standards have made it possible to implement sophisticated techniques, running in real-time. However, developing on the GPU has significant drawback of lacking debugging capabilities.

In order to alleviate this, a number of tools were proposed. Most of them operate on a very low level, which sometimes is inadequate due to complexity of the algorithm. Thus, debugging provides no high-level description of the underlying algorithm. In order to address these challenges, developers of an algorithm manually create high-level illustrations, reflecting the concept of the algorithm. These illustrations very clearly explain the algorithm, however, they have limitations. First, they are static and do not show cause and effect. Second, they are rendered manually, which requires a lot of work from a scientist or an illustrator. Finally, they are *a posteriori* with respect to the problem. This means that a researcher first understands the problem and then creates such an illustration. In our work we reverse this order, enabling the viewer to understand the algorithm features *from* the illustration. Visualization of the data structures the algorithm uses may provide considerable help in the further development and refinement of the algorithm.

In this paper, we contribute a framework, which visualizes the workflow of an algorithm by interactive exploration of its parameter space. The framework is integrated in the visualization software VolumeShop [BG05]. The proposed solution enables flexible visualization of data at different levels of abstraction: an entire image, a single fragment and on a sub-fragment level. Currently there is no framework which offers the possibility to interactively and visually understand the workflow of an algorithm. We call this paradigm *meta visualization*.

The rest of this paper is organized as follows. In Section 2 we discuss related literature on shader debugging and visualization. In Section 3 an overview of the framework is given. Different visualizations that we created for the framework are described in Section 4. We present the hierarchical classification with examples for each abstraction level in Section 5. Implementation details are presented in Section 6. Results of the approach are given in Section 7. We conclude, discuss limitations of the framework and outline directions for future research in Section 8.

# 2 Related Work

In recent years a lot of work has been conducted in the fields of shader debugging and software visualization. A number of systems assist developing proper visualizations or use visualization for guiding the creation of complex systems. Below we provide an overview of the recent research.

Laffra and Malhotra [LM94] implemented a system which enables the developer to debug C++ code, considering object-oriented specifics and typical bugs. Features like number of class instances, diagrams and related statistics are visualized. The topic of debugging visualization software was developed by Crossno and Angel [CR99]. Using the case study of a particle system they show, how different simple techniques (like color coding) are used for debugging. Duca et al. [DNB*05] introduced an SQL-like language which allows queries (e.g., *SELECT* all fragments, which correspond to a given primitive). Their approach is very powerful in retrieving the desired data from the OpenGL pipeline. However, there are certain limitations to it. This approach is not interactive (the video, accompanying the paper, shows performance of one frame per several seconds) and the visualizations are fixed and cannot be changed in a flexible way. Most recently, Laramee [Lar10] came up with a set of guidelines, targeted specifically for debugging visualization software. He proposes using customized code, visualizing the data of interest for visualization algorithms. "Traditional tools for debugging are of limited use because they de-couple the information they report from the spatio-temporal domain in which unexpected problems occur". Our meta visualization framework works interactively. Furthermore, it offers a versatile control over a visualization algorithm. A user is capable to visualize a certain parameter with various available visualization techniques or create his own one and plug it to the framework.

In recent years GPU performance has increased and as a consequence shader programs have become more complex. However, debugging of shaders remains problematic. There were no dedicated *debuggers for shaders*, so a developer was able to understand a shader function using a hit-and-miss method only. Trebilco [Tre06] proposes a tool, alleviating shader-intensive application development, called "GLIntercept". GLIntercept enables observation of specific shaders used for a particular part of the scene. A user can then edit shaders on-the-fly without the need of recompiling the application. Ehrath [Ehr06] presents a tool specifically suited for shader debugging. His "Print Shader" is able to output text values for the selected screen locations. Thus, actual variable values can be shown. Hilgart [Hil06] described a rewriting method that enables stepping through a GLSL shader program. The results were shown simultaneously for all fragments. Strengert et al. implemented [SKE07] glslDevil, a high-level shader debugging tool that uses hardware-accelerated rendering rather than software emulation. It also supports performance analysis, but requires user interaction for conditionals and loops. The glslDevil can produce simple visualizations such as color-coding. Though it is fairly powerful, it cannot provide interactive performance, as it suspends the program execution after every OpenGL call. Among commercial applications, there are tools such as gDEBugger [Gra04], PIX for Windows [Mic07] and the latest NDIVIA Parallel Nsight [NVI10]. The gDEBugger allows profiling an OpenGL application, finding its bottlenecks and showing OpenGL calls. However, it does not feature any means for reading a variable value in the shader code. PIX goes beyond this functionality, making it possible to debug a particular fragment [Mic07]. However, the debugging process is very low-level, software-emulated and, therefore, slow. A user cannot change a parameter value and immediately see the intermediate data and how it affects the final result. Nsight allows step-by-step debugging of high-level code and setting breakpoints [NVI10]. However, it is not interactive and does not provide any visualizations of the algorithm itself, except for performance graphs. Thus, so far, no method or framework was proposed to perform real-time visualization of display algorithms, utilizing shaders. We address this challenge by developing interactive extraction of shader parameters. The flexible mechanism is then used to provide these parameters as the inputs to visualization modules.

As our framework facilitates better algorithm understanding, we review related literature in the *software visualization* field. Diehl [Die07] gives an overview of a number of techniques and applications in this area. Since our work relates to interactive algorithm visualization, we focus on the *dynamic program visualization* domain. One of the first seminal works in the *algorithm animation* field was the *Sorting out Sorting* video [Bae81] which visualizes sorting algorithms on a value-position grid. Later, there were a number of noticeable systems created. One line of systems was founded with BALSA [BS84]. It introduces the concept of *interesting events*. At certain points of program execution a user inserts method calls, which send the desired variable values (as well as the event type). This concept cannot be re-utilized for our problems, as one cannot send an event directly from shader code. The second line of systems originates from TANGO [Sta90]. It im-

plements the *path-transition paradigm* for a continuous animation. The TANGO system was *postmortem*, which means that "the algorithm is executed, a trace or animation plan is produced and later visualized by a separate component" [Die07]. This paradigm is incompatible with display algorithms for two reasons. First, a display algorithm usually produces a huge amount of data per frame, which is not manageable in a reasonable period of time. Second, an interactive scenario for visualization is much more preferable, so that a user may inspect several points of interest during application run time and obtain the corresponding visualizations immediately. As an alternative to the interesting event concept, the LEONARDO [DF] framework introduces visualization by declarations. These declarations are comments, which are parsed by a virtual machine and trigger visualizations of variables values coupled with the algorithm. While we also make use of the parsed comments approach, our framework extends this concept through separation of an algorithm and its visualization, which enables using different visualizations for the same algorithm without additional coding. Another group of methods addresses the problem of visual debugging. Data Display Debugger [ZL96] allows the user to inspect the program state. For instance, it is capable of unfolding a list structure step-by-step. Traversal-based visualization [KA98], instead, unfolds all the data structures and visualizes it, using visualization rules. Zimmermann and Zeller introduce memory graphs [ZZ01] which show the memory state of a program by associating nodes with memory contents and arrows with references between them. This idea was adapted for object-oriented environments by de Pauw and Sevitsky who use reference patterns [PS99]. However, all these approaches are tailored for debugging data structures like lists or trees. A great variety of structures are not widely used in most of display algorithms, except for ray tracing. There are also attempts to use visualization for an error prediction. The goal is to detect a possible erroneous code, and techniques are based on the assumption that when the program runs with an error, the most frequently used parts probably contain a bug. The seminal system in this field is Tarantula [JHS02] developed by Jones et al. However, this system proved to be not very effective as it fails in the cases of loops and recursive functions.

A lot of systems *assist creation of visualizations*. Freire et al. [FSC*06] created VisTrails, which captures provenance for data and workflows to ensure reproducibility of results. They build a tree for the decisions, made during data exploration to figure out which parameters or workflows can be changed. In their follow-up work, Koop et al. [KSC*08] implement the VisComplete system which provides a user with a capability of auto-completing VTK pipelines, using a database of previously created pipelines. Santos et al. [SLA*09] proposed a tool for creating an application from a medley of visualization pipelines. Bruckner and Möller [BM10] presented a tool for alleviating navigation in the parameter space for artists, performing a multiparameter effects simulation. The success of above-mentioned systems show that visualization can be a very powerful tool in the development of other visualizations. However, these systems solve the problems in very particular cases. Instead, we propose a system which is generic enough to make visualizations for arbitrary display algorithms implemented using shaders.

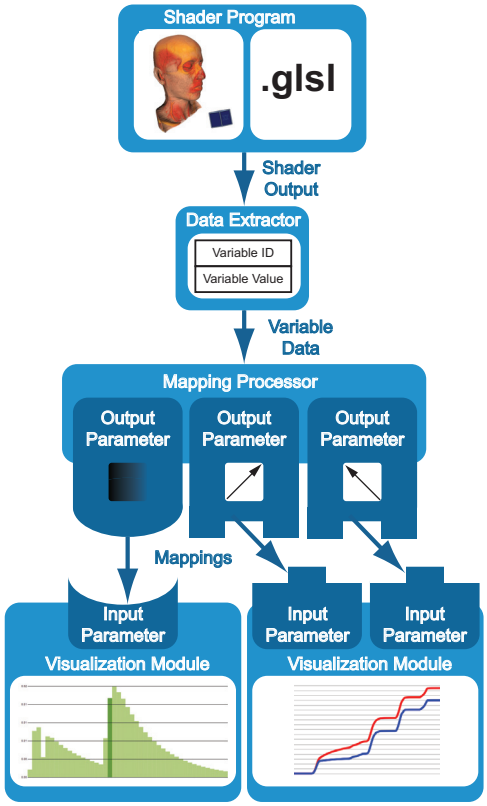


Figure 1: Framework overview. From top to bottom: the data extractor module acquires information from the shader, the mapping processor creates outputs, which are then fed to the appropriate inputs of different visualization modules.

## 3 Overview

This section describes the core of our framework: the *parameter-visualization* architecture. We first give an

overview for the framework and then in detail discuss its modules. Afterwards, we discuss the methods we introduce: overlay visualization and feedback visualization.

## 3.1 Framework Structure

Figure 1 shows an overview of the framework. The framework takes a *shader* file, which contains an algorithm implementation as an input. A user needs to specify which variables and parameters they would like to visualize.

First, at each frame, these data are extracted without a serious performance penalty. This is done by the *data extractor*, which retrieves this information from the shader and parses the layout of the data. The output of the data extractor is a data structure which contains the variables and parameter values computed for the current frame.

Second, the data is visualized. The *mapping processor* performs mapping of one or more variables or parameters to a *visualization module*. These modules together with the module which implements the algorithm form a multiple linked views visualization. Whenever the algorithm data change or a user interacts with a visualization module, all other modules are updated simultaneously, so they are preserved in a consistent state. Thus, display algorithm parameters are visualized by another visualization algorithm implementing the meta visualization.

## 3.2 Parameter Extraction from a Shader Program

Data from GPU algorithms cannot be obtained straightforwardly. A shader is unable to transfer data structures, other than textures (or images), to the CPU memory. This is why we propose a customized technique for transferring this data, described below.

We localize selection of the parameter in the shader file itself. This is done via adding annotations. The following code snippet takes as an input the ray coordinates for the current fragment and marches through it. We output variables `vecSampleValue`, `vecSampleColor` and `count`:

```
for (int count = 0; count < volumeDepth; count++)
{
  ...
  //Accumulate the ray
  ...
  vecSampleValue = GetValue(...);
  //<output vecSampleValue>
  vecSampleColor = GetColor(...);
  //<output vecSampleColor>
  if (vecRayColor.a >= earlyRayTerminationCoeff)
    break;
}
//<global output count>
```

Here the `vecSampleValue` and `vecSampleColor` are output for a user-selected fragment, while the `count` variable is output for an entire image (note the `global` modifier).

When the algorithm is compiled with a regular shader compiler (without meta visualization support), the comments are fully transparent to the compiler and do not create any additional overhead. If a variable changes throughout the program, it is possible to output it multiple times. If a variable shows up in a loop, output is automatically created for each iteration and the user does not need to perform any extra actions. The framework allows outputting an arbitrary number of variables per shader. The results are output to an image (texture). Each variable output is associated with a unique ID. Those are used afterwards during the data extraction phase.

The framework environment performs a preprocessing of the shader code. At the very beginning of the shader, a counter is initialized. This counter shows a position in the image, where the values are written. Each comment is replaced with the write instructions, which output the variable value and its ID to the texture in the position, defined by the counter. The counter is increased afterwards. After rendering is done, this image is read from the GPU to the CPU memory. It is then passed to the data extractor.

## 3.3 Data Extractor

The data extractor parses the shader output and then performs a mapping of the extracted data to the parameters. Conditional and loop operators may enable a variable to appear in the output several times or never. The data extractor extracts data from the shader output and stores values corresponding to the different IDs separately.

After the data are successfully extracted, they should be mapped to the corresponding variable name. This is done via back-projecting of the IDs to the names of the variables. The ID is read from a lookup-table, created during shader parsing, and the corresponding name is then associated with the extracted data. These operations create a set of data structures, which consist of variable names and their values, evaluated during the shader execution. This set is then passed to the mapping processor, which assigns appropriate visualizations.

### 3.4 Mapping Processor

After variable or parameter data were extracted from the shader, they need to be visualized. One possible solution is to statically link algorithms and the visualizations. However, this would result prevent of reusing these visualizations for other algorithms. Therefore, we de-couple an algorithm from its visualization by introducing a mapping processor and independent visualization modules. The mapping processor maps the parameters acquired at the previous step to the visualizations. In order to be able to receive these parameters, it needs to conform to the interface of the data structures set produced by the data extractor.

The mapping processor provides visualization modules with an interface to the extracted data. It allows passing the same variable data to different visualization modules avoiding unnecessary data duplication. The mapping processor is also responsible for visualization updates. Whenever a variable changes its value, the mapping processor sends a message to all visualizations currently using this variable.

### 3.5 Visualization Modules

In our framework visualizations are implemented as independent modules. This enables reusing of visualizations for newly implemented algorithms. If existing modules are not expressive enough for visualization of a particular algorithm, a user can simply implement new visualization modules and add them to the framework.

Each visualization module shares a common interface. This interface allows the module to be registered by the mapping processor to receive updates. Besides, the visualization module declares its parameter inputs used by the mapping processor. The module also provides GUI for mapping variables and the specification of visualization parameters (e.g., colors and captions).

### 3.6 Feedback and Overlay Visualization

While visualization generates, in essence, output data, it can also generate input data for other visualizations. For example, if a bar chart visualization is used, a user might be interested in, which particular datum corresponds to a certain bar. To achieve this, we allow visualization modules to send a *feedback* to the mapping processor. Figure 2 shows this concept for the bar chart visualization sample. The index of the selected bar is transferred back to the mapping processor which then feeds it to another visualization module. This module can then visualize any specific information related to the data corresponding to this particular bar.

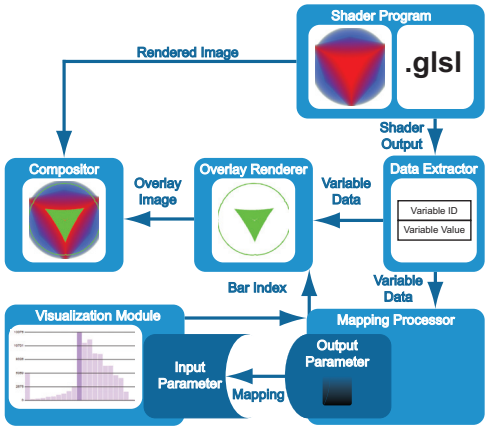


Figure 2: A sample pipeline for feedback and overlay visualization.

Until now we have been separating a display algorithm and its visualization. They are shown in different views and have not been spatially linked. However, in certain situations, it can be convenient to merge an algorithm and its visualization. For example, one might be interested in which fragments of an image possess certain characteristics. Therefore, we introduce an *overlay* visualization. Figure 2 presents an example of the pipeline for overlay visualization. The bar chart visualization module transfers the data to the mapping processor corresponding to the selected bar. These data are then passed to the overlay renderer. Finally, a compositor fuses them together with the output of the renderer that is visualized.

## 4 Visualization Modules

In this section we discuss different visualizations and the motivation behind them. We do not focus on creating a comprehensive collection of different visualization algorithms. Instead, we implement a selection, which show the main features of the framework.

We use a synthetic scene that consists of an intersecting ball and a box in order to demonstrate visualizations. We will use visualization to analyze a simple volume ray-casting algorithm presented in the code snippet in Section 3.2.

### 4.1 Bar Chart

**Input:** a single variable, inside or outside a loop.

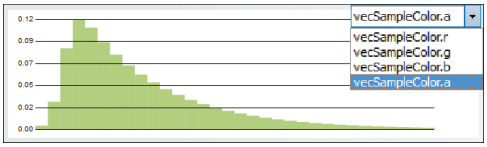


Figure 3: Bar chart visualization and accompanying GUI. Notice that one can link any variable from the shader output to this visualization.

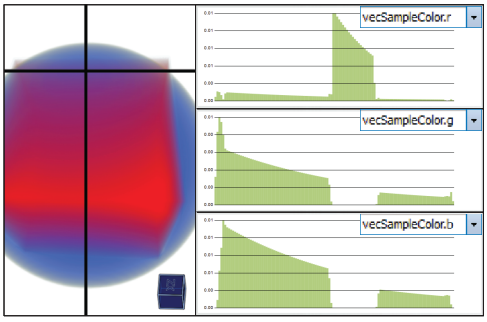


Figure 4: Three bar chart diagrams, from top to bottom, assigned with the parameters `vecSampleColor.r`, `vecSampleColor.g` and `vecSampleColor.b`, respectively. The black cross shows the fragment for which the ray profiles are shown.

A bar chart is a traditional technique for visualization. It is able to present information in a very accessible way. It also has a major advantage over a graph visualization: when the data are discrete, it shows every individual datum clearly.

A bar chart can take one input parameter. All the values from this parameter (e.g., if it was assigned several times in a loop) are fed to the bar chart. Figure 3 shows a simple bar chart. If a user selects a `vecSampleColor.a`, they will get an opacity ray profile.

We demonstrate how a single visualization module can be reused for visualization of different parameters. In this example, it is interesting to know how the particular materials are distributed along the ray. To achieve that, we can reuse the above-mentioned bar chart visualization. We create three independent bar charts and assign a color channel per each of them shown in Figure 4.

### 4.2 Graph Visualization

**Input:** 2 variables (one variable per axis).

Being a traditional visualization, graph visualization is of a great use. If data convey some geometrical characteristics (e.g., surface height or ray angle) a graph can show important spatial information.

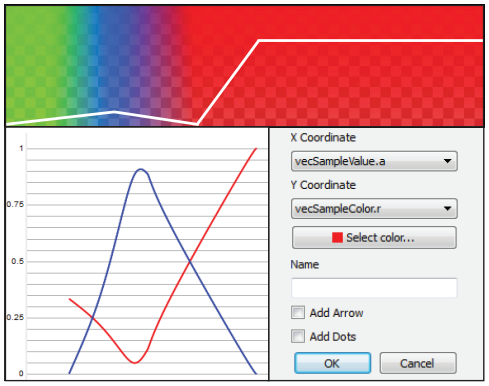


Figure 5: Graph visualization of a sample value against sample colors. On top the transfer function is shown. At the bottom left the graph visualization is shown. At the bottom right the visualization module GUI for adding a new graph is presented. This visualization is made for the same fragment and for the same scene as in Figure 4.

We implemented a graph visualization where a user can add an arbitrary number of graphs to visualize. A user selects one parameter to be an abscissa (or an array of them – in the case of a loop) and the other – to be an ordinate of the graph, as well as a color and a caption. A user is also able to attach an arrow to the end of the graph and adding key points (dots) for feedback visualization.

Figure 5 shows an example of the graph visualization. In this example a user visualizes a dependency between a density of a fragment (`vecSampleValue.a`) and its color (`vecSampleColor.r`, `vecSampleColor.b`). The transfer function on top is given as a reference.

## 5 Display Unit Hierarchy

Shneiderman [Shn96] introduced the visual information-seeking mantra: “Overview first, zoom and filter, then details on demand”. Similar concepts can be applied to the exploration of a display algorithm. First a developer examines the overall image. Then, they take a look at specific fragments. For example, for a soft shadowing algorithm, one would inspect fragments at a border of a shadow carefully. Finally, for a complex algorithm, it may be worthwhile to go down one more level to the sub-fragment level. We describe the ability to of explore these levels offered by our framework.

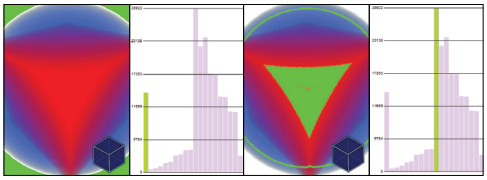

Figure 6: Using overlay feedback visualization for ray length visualization. From left to right, ray length is increasing. The overlay is rendered in green.

## 5.1 Image Level

At the image level one might want to output and analyze a variable value for a whole image. To do so, a user creates a *global* output. For global variables we limit ourselves to one output per fragment. This is done in order to keep the output to a reasonable size and easily interpretable.

After this output is received, a user can visualize them using visualization modules of the framework. It might be interesting to get a histogram of ray lengths all around the image and understand where the points with certain ray lengths are. To support this we use feedback visualization for transmitting the index of the hovered histogram bar. Fragments which correspond to this histogram bar are then displayed using an overlay visualization. The `count` variable contains the number of ray loop iterations or ray length. Figure 6 shows an overlay feedback visualization for the ray lengths histogram. One can inspect, for instance, which fragments correspond to the peak on the histogram.

## 5.2 Fragment Level

After the *overview* has been provided, it is time to *zoom and filter*. A classic example is a ray profile for raycasting algorithms. This provides a great cue for understanding an algorithm. To output the value of a single fragment one uses shader comments.

A user needs to specify (via the GUI) a position of a specific fragment which he wishes to visualize This information is passed to the data extractor as shown in Figure 7. Unlike the image level, at the fragment level a value can be output several times, e.g., in a loop. Regardless the actual number of outputs, the outputs are identified automatically by the data extractor.

## 5.3 Sub-fragment Level

Complex display algorithms perform many operations on a single fragment. Therefore, one might be interested in going one level down and seeing more detail.

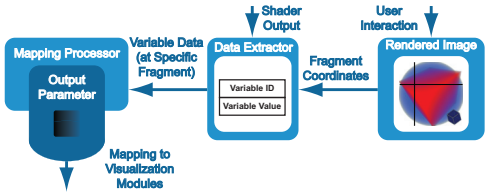


Figure 7: Fragment level visualization pipeline. Values, corresponding to a user-defined fragment are extracted by the data extractor and passed to the mapping processor.

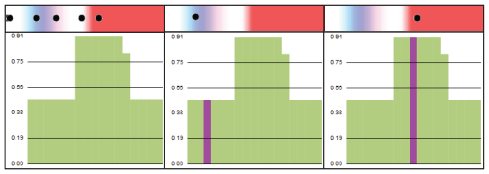

Figure 8: Transfer function visualization at the sub-fragment level and ray profile visualization (bar chart). Texture fetches are shown with black dots for the whole ray (left) and at concrete voxels (middle and right). The user-selected bars are highlighted with violet. This visualization is given for the same fragment and for the same scene, as in Figure 4.

One particularly interesting feature at a sub-fragment level is texture fetching. It is important to know how many fetches are performed, where the texture is fetched and which iterations (if many fetches are done) they correspond to. For example, artifacts in the *percentage-closer soft shadow* algorithm [Fer05] occur due to large distances between shadow samples. Visualization of those samples helps understand the problem.

We automatically parse all texture fetches in the shader file. First, each texture definition is analyzed and the texture dimensionality is acquired. This is needed for correct output generation. Then the shader file is searched for texture fetching commands and each of them is output. We write the texture ID and the fetch coordinates for each texture fetch. We can also link a texture fetch with a variable, which uses results of this fetch. Thus, a user can selectively visualize only the texture fetches which correspond to the variable value at the concrete iteration.

Figure 8 shows an example of a sub-fragment texture fetch visualization. We decided to visualize texture fetches (shown as black ellipses) which are made from a texture that stores a transfer function. This visualization is linked with ray profile visualization (bar chart), which enables the user to browse texture fetches on the sub-fragment level. Each bar corresponds to a single it-

eration of the ray-casting algorithm. When a user hovers on a bar in the bar chart, only the texture fetch made at the same iteration is shown.

# 6 Implementation

The proposed framework has been implemented on the basis of the VolumeShop system. Modules are implemented as plugins to VolumeShop. For rendering and GUI programming we used C++ and OpenGL (GLSL is used as a shading language). Visualization modules were implemented in Protovis [HB10]. These visualizations were then rendered as a web-page, using Qt WebKit [Nok08].

The shader output is performed using an `EXT_shader_image_load_store` extension available in OpenGL 4.0. We use its ability to make an arbitrary number of memory writes from a single fragment shader. Separate textures are created for global outputs, local outputs and texture fetches.

Output commands in the shader code are parsed with a simple preprocessor before the shader compilation and then replaced with memory writes. If a user does not use the `ShaderDebugger` class these commands simply remain as comments and do not affect the rendering process at all.

The `ShaderDebugger` class creates a so-called *shader resource*: an entity which can be shared among different modules. This shader resource is then linked to all modules participating in the visualization and rendering process. This resource serves as means for cooperation. First, the data created by one module (i.e., outputs from the mapping processor) and used by another (in this case, some visualization module) are transmitted through this resource. This enables encapsulating a sender and a receiver from each other. Second, signals of the shader resource are used for module synchronization. For example, when the rendering module updates the frame it triggers a corresponding event to the visualization module so it can update its visualization accordingly. And vice versa, when a user interacts with feedback visualization the visualization plugin fires a redraw event for an overlay visualization module. The overlay is updated accordingly.

The multiple linked view capability is supported natively. Because modules operate as independent plugins a user can add multiple visualizations for an algorithm. These modules are synchronized in a consistent state through the shader resource described above.

The framework works interactively for the presented algorithms. All resource and variable linking operations are performed via the GUI and no additional programming is required.

# 7 Results

In this section we present results of a case study for the proposed framework. We demonstrate our technique on two algorithms: for polygonal rendering and for volume rendering. First we present our technique on the parallax occlusion mapping algorithm. We took as a basis its implementation from the RenderMonkey IDE [Adv08] and integrated it to VolumeShop as a plugin. The second algorithm we selected is MIDA proposed by Bruckner and Gröller [BG09]. It has been already implemented in VolumeShop as a separate plugin.

In the case of the parallax occlusion mapping algorithm we used the framework to find and eliminate an error in the algorithm implementation and to provide a visual cue for the aliasing problem. For both algorithms we were able to produce illustrations, resembling illustrations from the original papers which helped users understand the algorithm. These illustrations are based on the actual data and are completely interactive.

## 7.1 Parallax Occlusion Mapping

In this section we describe the visualizations we created to resolve the issues with parallax occlusion mapping using the meta visualization framework. We will first give a short overview on the parallax occlusion mapping algorithm itself, while further details can be found in [Tat06].

Many objects in real life have tiny details on the surface. A good example is a cobblestone road. If we try to simulate each cobble with a set of polygons, it will be prohibitively expensive. Instead, we can use different texture mapping approaches to model it.

A straightforward approach is just to apply a color texture. However it fails to convey the material of the surface properly especially when it is specular. Bump mapping dramatically improves rendering quality by per-pixel normal permutation. Per-pixel normals are acquired by height map differentiation. However, bump mapping does not convey a parallax: an effect of self-obstruction of a surface relief. This can be simulated by shifting texture coordinates based on the local height value. For accurate evaluation of the texture coordinates shift local ray casting is used as shown in Figure 9 (top). Figure 10 demonstrate the same scene, rendered with all of these 3 techniques.

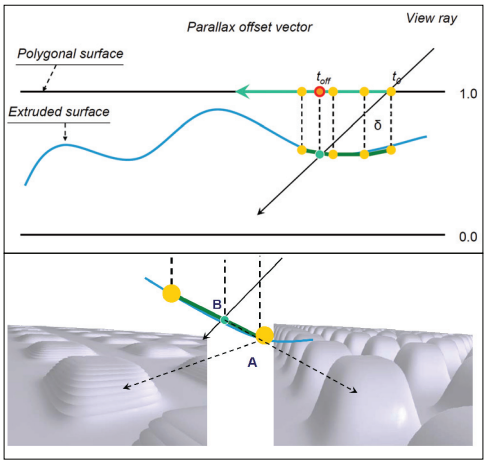


Figure 9: Parallax mapping illustrations from the presentation: the intersection of the viewing ray with the surface and the corresponding texture coordinate offset (top), the aliasing problem, occurring because due to undersampling (bottom). Image courtesy of Natalya Tatarchuk.

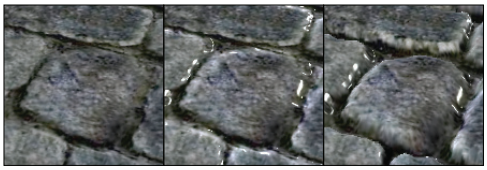

Figure 10: Texture algorithms for simulating surface details: texture mapping (left), bump mapping (middle), and parallax occlusion mapping (right).

When we first ported the algorithm to VolumeShop the problem shown in Figure 11 (left) was found. This artifact (shown in the red close-up) occurs at certain angles resulting in discontinuities in the final image. In order to find its cause we took a look on the corresponding source code. It takes the normal map, the height map, and the current viewing ray as an input.

```
while (nStepIndex < nNumSteps)
{
  vecTexCurrentOffset -= vecTexOffsetPerStep;
  fCurrHeight = texture(samNormalMap,
        vecTexCurrentOffset).a;
  fCurrentBound -= fStepSize;
  if (fCurrHeight > fCurrentBound)
    break;
  else
    nStepIndex++;
}
```

Here, `nNumSteps` represents the maximum number of allowed steps, `nStepIndex` stands for the current number of the step, `fCurrHeight` is the current *y* coordinate of the viewing ray, while `fCurrentBound` is the current height of the surface (which is taken from the a-component of the normal map texture where the height map is stored), and `fStepSize` is the viewing ray *y* coordinate increment per step (which is dependent on the angle between the ray and the surface).

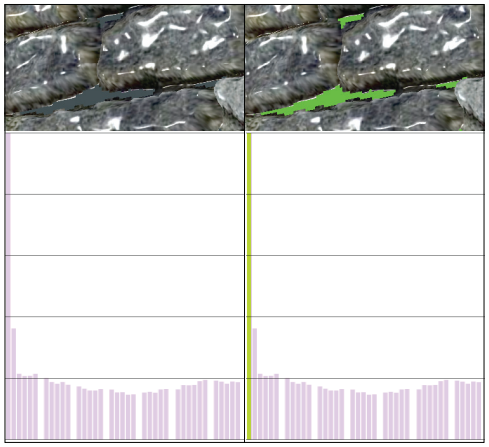

Figure 11: Image level visualization for parallax occlusion mapping. Global output of the `nNumSteps - nStepIndex` expression. The user hovers over a corresponding bar to obtain an overlay rendering. Notice the solid filled gray area on the close-up which stays out from the nicely lit valley.

This artifact might be somehow linked with the fact that the iterative process has not ended properly. If we were able to see which fragments have run the loop until the very end that might give us a clue. Let us create an output for the expression `nNumSteps - nStepIndex`. The smaller this value is, the earlier the ray has terminated. Figure 11 shows the image level visualization for this expression. As one can observe, the areas with flat discontinued shading correspond to the case when the loop runs to the very end. This means that the actual intersection is not found for these pixels, because one more step is missing. One of the possible bug fixes would be to replace the loop condition `nStepIndex < nNumSteps` with `nStepIndex < nNumSteps + 1`.

Another issue, which occurs for parallax occlusion mapping at acute angles is aliasing. Aliasing occurs due to undersampling (see Figure 9 (bottom)), but it is difficult to understand it at first. Figure 12 shows a visualization for parallax occlusion mapping, which reveals the reasons for aliasing. This visualization mimics that from Figure 9, but in contrast, it is interactive and matches the actual user-defined data.

In order to obtain a deeper algorithm understanding, one can view the sub-fragment level. For example, to understand why the extruded surface plot has such a shape it is

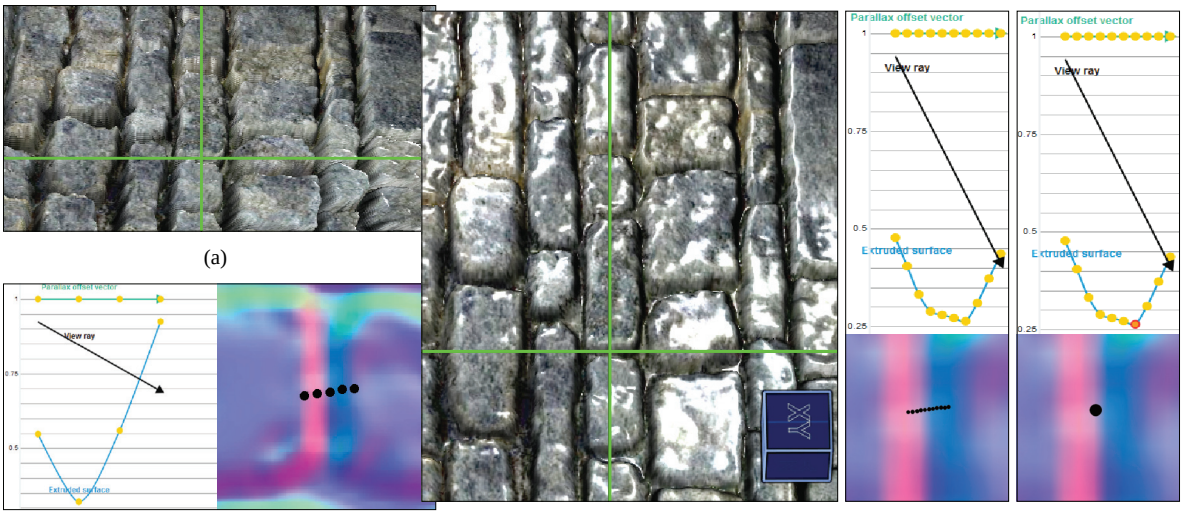


Figure 12: Aliasing at an acute angle in parallax occlusion mapping. Notice the differences in rendering for the same surface under different angles: a) aliasing occurs – notice how inaccurate the solution on the right is (see (b)), c) sampling is dence enough such that no aliasing occurs (see (d)). The green crosses show the fragments being visualized. The image (e) shows the texture fetch corresponding to the user-selected variable output.

worthwhile to look at how the texture has been sampled. Figure 12 shows such a visualization featuring *multiple linked views*. As can be observed, association of the texture fetch with the corresponding iteration makes it possible to browse through the texture fetches corresponding to the concrete iterations. Texture fetch visualization provides a user with an important visual cue. It shows the actual parts of the texture which contribute to the fragment. For example in Figure 12 (b) we can see that the extruded surface is highly curved because it corresponds to a part between two bricks in the normal map.

### 7.2 Maximum Intensity Difference Accumulation

We have also tested our framework on a volume visualization algorithm. We have selected the maximum intensity difference accumulation algorithm (MIDA) [BG09] algorithm, as it changes the opacity value along the ray in a peculiar way.

MIDA is a volume rendering ray-casting algorithm. It strives to combine advantages of direct volume rendering (DVR) and the maximum intensity projection (MIP). Essentially it works like DVR, but when a new local maximum is achieved along the ray the accumulated opacity is decreased, so this maximum affects the final rendering result.

Figure 13 paper shows how the data value, the intensity and the opacity are modified by MIDA algorithm at local maxima in order to reveal inner structures. We strived to achieve a similar visualization to create a storytelling image which reflects the actual way the algorithm works. To fulfill this, we output parameters from the shader: the accumulated opacity, the intensity, the data value, and the distance, travelled by the ray, in order to parameterize the plot. Notice that due to the opacity decrease at local maxima, the hidden high-intensity details, like teeth, become visible.

## 8 Conclusion and Future Work

In this paper we propose the framework for *meta visualization*: visualization of display algorithms. We also demonstrate the apparatus for interactive extraction of arbitrary variables: independent of if they are in a loop or not. In addition a hierarchical classification is introduced for visualization of these data. It can be performed globally for a complete image, for one concrete fragment, or at the sub-fragment level. This framework works interactively and allows a user to change visualization parameters on-the-fly, taking advantage of multiple linked views. Finally, we demonstrate how these methods can be applied to two algorithms: one for polygonal and one for volume-rendering. We showed

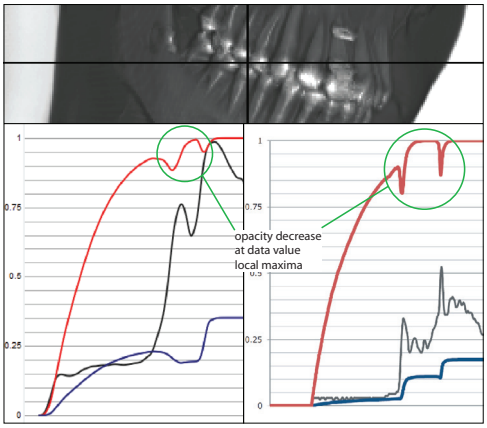


Figure 13: Ray profiles for MIDA. The image at the bottom left has been interactively generated by the proposed meta visualization framework for the scene, shown on top. The image at the bottom right has been taken from [BG09] (it relates to another scene). Image courtesy of Stefan Bruckner.

that illustrative visualizations that elevate display algorithms understanding can be easily generated interactively using our framework.

We make the system as generic as possible while preserving its usability. However, it has limitations. We currently perform visualization only for data in fragment shaders. It would be interesting to visualize data for other types of shaders (e.g., vertex and geometry shaders) especially for polygonal rendering algorithms. Second, the current implementation works as a set of VolumeShop plugins. A user needs to port his algorithm to this system. It would be much more convenient to use this framework as an independent library. We do not enable the use all outputs of a variable in a loop at the image level due to performance issues. This can be potentially helpful for complex volume rendering algorithms.

We have different directions for further development of our system. First, visualization for a shader code itself might be very helpful. For example, if a user selects a fragment, they should get the parts of the shader which are used for rendering of this fragment, highlighted. Second, more analysis can be done on the final result dependency from input data. For example, if an algorithm performs multiple memory fetches we would like to know which fetches are important and which are not so they could be omitted without a serious impact on the quality. Finally, higher-level semantics can be introduced. If we limit ourselves to the ray-tracing or ray-casting algorithm, the ray itself can be considered as a unit of hierarchy. Lower hierarchy levels in this case will be derivative rays (e.g., reflected, refracted and shadow rays).

## References

[Adv08] ADVANCED MICRO DEVICES INC.: *AMD RenderMonkey IDE Version 1.82*, 2008.

[Bae81] BAECKER R.: Sorting out sorting. 30 minute color film (developed with assistance of Dave Sherman, distributed by Morgan Kaufnamm, University of Toronto), 1981.

[BG05] BRUCKNER S., GRÖLLER M. E.: VolumeShop: An interactive system for direct volume illustration. In *Proceedings of IEEE Visualization 2005* (October 2005), pp. 671–678.

[BG09] BRUCKNER S., GRÖLLER M. E.: Instant volume visualization using maximum intensity difference accumulation. *Computer Graphics Forum 28*, 3 (June 2009), 775–782.

[BM10] BRUCKNER S., MÖLLER T.: Result-driven exploration of simulation parameter spaces for visual effects design. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (October 2010), 1467–1475.

[BS84] BROWN M. H., SEDGEWICK R.: A system for algorithm animation. *SIGGRAPH Computer Graphics 18*, 3 (January 1984), 177–186.

[CR99] CROSSNO P., ROGERS D.: Visual debugging of visualization software: A case study for particle systems. In *Proceedings of the Conference on Visualization '99* (1999), pp. 417–420.

[DF] DEMETRESCU C., FINOCCHI I.: LEONARDO Webpage. http://www.dis.uniroma1.it/ demetres/Leonardo.

[Die07] DIEHL S.: *Software Visualization*. Springer-Verlag, 2007.

[DNB*05] DUCA N., NISKI K., BILODEAU J., BOLITHO M., CHEN Y., COHEN J.: A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics 24*, 3 (July 2005), 453–463.

[Ehr06] EHRATH A.: Print shader for debugging pixel shaders. In *ShaderX5: Advanced Rendering Techniques* (December 2006), Charles River Media, pp. 591–594.

[Fer05] FERNANDO R.: Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches* (2005).

[FSC*06] FREIRE J., SILVA C. T., CALLAHAN S. P., SANTOS E., SCHEIDEGGER C. E., VO H. T.: Managing rapidly-evolving scientific workflows. In *Proceedings of International Provenance and Annotation Workshop (IPAW)* (2006), pp. 10–18.

[Gra04] GRAPHIC REMEDY: gDEBugger. http://www.gremedy.com, 2004.

[HB10] HEER J., BOSTOCK M.: Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (November 2010), 1149–1156.

[Hil06] HILGART M.: Step-through debugging of GLSL shaders. Technical Report, 2006.

[JHS02] JONES J. A., HARROLD M. J., STASKO J.: Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (2002), pp. 467–477.

[KA98] KORN J., APPEL A.: Traversal-based visualization of data structures. *IEEE Symposium on Information Visualization* (1998).

[KSC*08] KOOP D., SCHEIDEGGER C. E., CALLAHAN S. P., FREIRE J., SILVA C. T.: VisComplete: Automating suggestions for visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (November/December 2008), 1691–1698.

[Lar10] LARAMEE R. S.: Using visualization to debug visualization software. *IEEE Computer Graphics and Applications 30*, 6 (November/December 2010), 67–73.

[LM94] LAFFRA C., MALHOTRA A.: HotWire: A visual debugger for C++. In *Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference* (1994), vol. 6, p. 7.

[Mic07] MICROSOFT CORPORATION: PIX: The Direct3D profiling and debugging tool. DirectX 10 SDK, 2007.

[Nok08] NOKIA CORPORATION: Qt features for hybrid web/native application development. White Paper, 2008.

[NVI10] NVIDIA CORPORATION: NVIDIA Parallel Nsight: Debugging massively parallel applications. In *GPU Technology Conference* (September 2010).

[PS99] PAUW W. D., SEVITSKY G.: Visualizing reference patterns for solving memory leaks in Java. In *Proceedings of European Symposium on Object-Oriented Programming* (1999), pp. 116–134.

[Shn96] SHNEIDERMAN B.: The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages* (1996), pp. 336–343.

[SKE07] STRENGERT M., KLEIN T., ERTL T.: A hardware-aware debugger for the OpenGL shading language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2007), pp. 81–88.

[SLA*09] SANTOS E., LINS L., AHRENS J. P., FREIRE J., SILVA C. T.: VISMASHUP: Streamlining the creation of custom visualization applications. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (November/December 2009), 1539–1546.

[Sta90] STASKO J. T.: TANGO: A framework and system for algorithm animation. *IEEE Computer 23*, 9 (1990), 27–39.

[Tat06] TATARCHUK N.: Dynamic parallax occlusion mapping with approximate soft shadows. SIGGRAPH Š06: ACM SIGGRAPH 2006 Courses, 2006.

[Tre06] TREBILCO D.: GLSL shader debugging with GLIntercept. In *ShaderX4: Advanced Rendering Techniques* (January 2006), Charles River Media, pp. 533–539.

[ZL96] ZELLER A., LÜTKEHAUS D.: DDD–a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices 31* (January 1996), 22–27.

[ZZ01] ZIMMERMANN T., ZELLER A.: Visualizing memory graphs. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures* (May 2001), pp. 191–204.