Matthias Raspe · Guido Lorenz · Stephan Palmer

Hierarchical and Object-Oriented GPU Programming

Abstract Using programmable graphics hardware in order to solve both computer graphics and non-graphics tasks has become commonplace in recent years. High level shading languages have greatly simplified the effort of writing programs for graphics processors, and dedicated programming interfaces proposed recently by major hardware vendors abstract from computer graphics details. However, algorithms that strive for solving real-life problems while maintaining direct visualization capabilities are rather complex and require much more than individual shader components.

In this work, we propose a framework for efficiently integrating programming resources of both GPU and host applications. We therefore introduce a hierarchical approach to structure GPU and CPU program components with additional data-driven control. Transparently shared parameters and computation results enable flexible communication between different computation steps, no matter if the computation is performed on the host or the graphics hardware. We illustrate a set of classes facilitating the proposed functionality, while retaining the full design and programming capabilities from object-oriented programming languages. These approaches are supplemented by techniques available in shader programming languages to provide even more flexibility. In order to demonstrate the potential of the approach, we present a reasonable scenario and discuss the framework's performance in terms of programming efforts and run time overhead.

Keywords GPU programming · Object-oriented programming · GPGPU

Matthias Raspe
Computer Graphics Working Group, Institute of Computational Visualistics, University of Koblenz-Landau
Universitätsstrasse 1, 56070 Koblenz, Germany
Tel.: +49-261-287-2794
Fax: +49-261-287-2735
E-mail: mraspe@uni-koblenz.de

Guido Lorenz and Stephan Palmer
Computer Graphics Working Group, Koblenz, Germany

1 Introduction

Using programmable graphics hardware for different kinds of computations has become very important during the past years and is the subject of ongoing research. With the rapid development – mainly driven by the game and entertainment industry – the raw performance of commodity graphics processors exceeds that of modern (multi-core) processors by far. Thus, they are also of interest for different non-graphics tasks, often referred to as "General Purpose GPU".

However, programming graphics hardware still requires an in-depth knowledge of computer graphics concepts and differs much from standard application programming in terms of flexibility, paradigm, development tools, etc. The former is alleviated by graphics hardware vendors' APIs like CUDA or CTM that regard the GPU simply as computing device. Introducing established software engineering concepts like object-oriented programming or design patterns to (GP)GPU programming has not been addressed in detail yet, however.

In this work we will propose two complementary approaches to improve shader programming. We will introduce a hierarchical representation for the building blocks of GPU functionality to allow for a flexible way of representing complete workflows. Our cross-platform system "Cascada" has been developed in the context of medical volume processing and visualization, but the approaches described are not limited to such applications. Furthermore, the handling of (offscreen) rendering components and their parameters has been implemented using an object-oriented approach and further software engineering techniques, enabling the flexible combination of GPU and CPU procedures.

The remainder of this paper is structured as follows. In section 2, we will give an overview of existing techniques in the context of our work. We will then explicate our concepts in section 3 and give details on both approaches proposed in this contribution. In addition, we will briefly outline our system and describe the con-

ISBN: 978-9944-5685-4-8

cepts using medical volume segmentation as an example. In section 5 we will present our results with respect to performance and programming efforts compared to plain graphics programming. We will conclude in the last section and propose directions for future work.

2 Related Work

In this section, we will outline related work in the field of generalizing GPU programming. We will also discuss the methods in comparison to our system with respect to shader programming and handling of parameters.

2.1 Shader programming

The programmability of the rapidly evolving graphics processors is today considered one of the key features of this hardware architecture. High-level languages like Cg or GLSL have become the de-facto standard for GPU programming. Although this already enables more flexible and rapid development of shaders, approaches aiming at an additional abstraction layer have been proposed. CgFX [4] and similar formats define self-contained shader units called *effects*. They bundle vertex and fragment shader code with render passes, states and parameters to describe a complete shading procedure that can be applied to any geometry. This approach has been extended by Eissele et al. [3] for their system working with data in image-space. While providing support for multiple render passes, the effects cannot be used to model iterative computations, especially under dynamic data-driven conditions.

McCool and others [7] have developed a meta-language concept to integrate shader functionality directly into the application code. However, we found the performance of the generated code is often inferior to manually written GLSL code due to redundancies or suboptimal constructs. Recently, Trapp et al. [9] have taken previous approaches further and propose a system for an automatic combination of high-level shader programs during run time. They split shader code into small fragments and augment them with semantic information. Thus, they are able to build, combine, and reuse different shaders efficiently.

Yet another approach is the representation of shader functionality as directed graph, with nodes representing shader fragments connected by input and output parameters [2,1]. This method has recently been extended by McGuire et al. by automatic parameter matching [8]. Finally, Kuck [6] has introduced object-oriented concepts to shader programming by means of proxy classes representing shaders and allow for problem-oriented rather than hardware-oriented programming.

2.2 Parameter handling

In the references above, little information is given about how parameters shared by the main program and the shader code – refered to as *uniform variables* in Cg and GLSL – can be handled efficiently with minimal programming effort. Modular approaches that concatenate shader fragments [2,1,8,9] focus on input and output parameter handling between individual shading modules, but do not address the question of how to incorporate shader parameters into an application.

The metaprogramming approach of McCool et al. [7] allows for a simple exchange of parameters, since both the main program and the shader code are written in C++. However, this cannot be applied to our solution, as we intend to use manually written and optimized GLSL code. With an object-oriented framework as proposed by Kuck [6], parameters can be exchanged between main program and shader code as members of common objects. Kuck stresses the fact that his approach is lightweight, which certainly applies to the fact that it has no additional run time costs. Yet, we found implementing a class shared by C++ and GLSL using his framework less intuitive than object-oriented programming in standard C++, due to the heavy use of templates and macros.

3 Our approach

In order to abstract from graphics programming details and allow for interchangeable GPU and CPU implementations, we propose two concepts. First, algorithms are represented hierarchically and can thus be (re-)used at different levels of granularity. Handling the shaders and, even more important, the different types of parameters in an object-oriented way is the second contribution in this work. Therefore, we have used several programming concepts and design patterns; the latter can all be found in [5].

3.1 Hierarchical rendering components

Representing algorithms hierarchically emphasizes the desired workflow rather than the hardware-specific task like render texture setup, shader initialization, etc. Our hierarchy is structured into three levels, as can be seen in figure 1. At the highest level are *sequences* that encapsulate procedures ranging for example from simple thresholding to more complex operations like region growing. Although one sequence can contain all stages of a workflow, it is advantageous to keep sequences as small as possible to reuse their functionality in different contexts, as exemplified in section 4. By utilizing the composite pattern, sequences are implemented as subclasses of `RenderComponent` and thus can contain sequences themselves, which allows for even more flexible and complex designs. Additional control is given by (de-)activating each component separately, which is possible during run time.

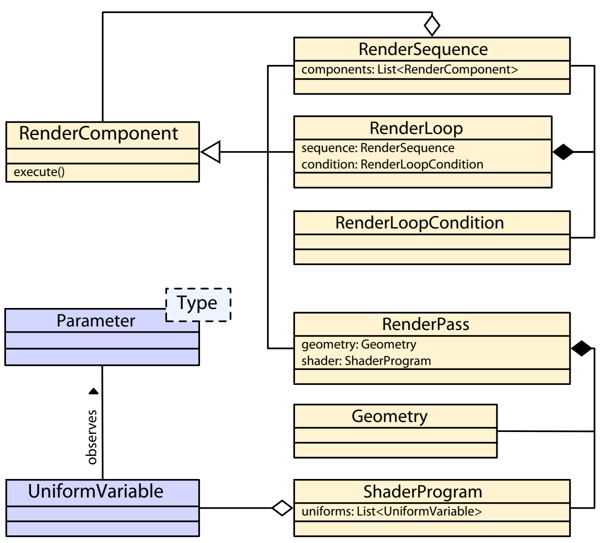

Fig. 1 Simplified class diagram illustrating the main rendering components (yellow) and uniform variables (blue).

At the next level are *passes* that draw geometry with assigned shader programs to a defined target. For convenience, we provide subclasses that already allow for offscreen rendering (i.e., screen-filling quadrilaterals to offscreen buffers), onscreen rendering of arbitrary geometry, volume raycasting, etc. These passes can resemble single operations or repeatedly run passes, controlled by a fixed number of iterations or until some condition is met (e.g., region growing has converged). Especially for iterative algorithms, we additionally provide a `RenderLoop` that wraps a render sequence being executed multiple times; this loop is controlled by some `RenderLoopCondition`. The output of one pass is then used as input to the subsequent pass by setting the texture parameters accordingly.

Finally, *shader programs* resemble objects containing GLSL programs. Unfortunately GLSL does not specify the use of files for its shaders yet, especially with include directives (such as in Cg; proposed for OpenGL 3.0). To circumvent this limitation in order to have small, reusable functionality, we integrated this feature by custom tools.

3.2 Parameters

A *sequence* as described in section 3.1 has a number of *parameters*, like the input data to process, and parameters specific to the algorithm it implements. Furthermore, there are parameters that are independent of individual sequences, like the current size of the rendering window. To distinguish these two classes of parameters, we refer to them as *local* and *global* parameters, respectively. Both are managed by a `ParameterSet`, one being part of the sequence, while the other one is made available globally using the singleton pattern. They hold instances of a `Parameter` class, that is templated to support different types of parameters, like single values, vectors, or textures.

To use these parameters in the shader code, both Cg and GLSL support the concept of *uniform variables.* In our framework, these uniform variables are represented by wrapper classes. Before a shader program is compiled, its source code is searched for uniform variable definitions and adequate `UniformVariable` objects are created automatically. In order to synchronize their values, these wrappers can then be attached to parameter objects using the observer pattern. This way, handling uniform variables reduces to connecting them to a parameter object once.

4 Application

After describing our approach from a conceptual point of view, we will now describe a real world scenario. Therefore, the example implements a segmentation sequence that is based on region growing. Input data is in our case medical volume data, but translates also to other data (e.g., 2D video frames), of course. As can be seen from the legend in figure 2, almost all parts of the sequence are performed on the GPU. In addition to the superior performance – especially for volume algorithms – the data can be visualized during run time with a negligible overhead, as the data being processed is already in video memory.

The whole workflow of the segmentation example is depicted in figure 2 as a (simplified) tree that is traversed in depth-first order. First, the preprocessing subsequence is executed by traversing all active components; in the example the bilateral filter has been deactivated an is thus omitted. Additional information like the histogram or gradients are computed on the following steps; note that CPU and GPU algorithms can be used interchangeably.

The region growing is implemented as a `RenderLoop` that is executed until convergence. In the example, some simple implementation is skipped leading directly to the computation of the input data's mean value. A reduction shader iterates until the texture is at minimum size and resembles the total mean value. Afterwards, some conditional dilation is performed that uses the mean value for classifying neighboring voxels. Then the difference between the current output and the output of the step before is computed, resulting in the number of fragments determined via occlusion queries (as optional property of `GLRenderPass`). If this number is zero, the region growing has converged (as iteration n and $n-1$ do not differ)

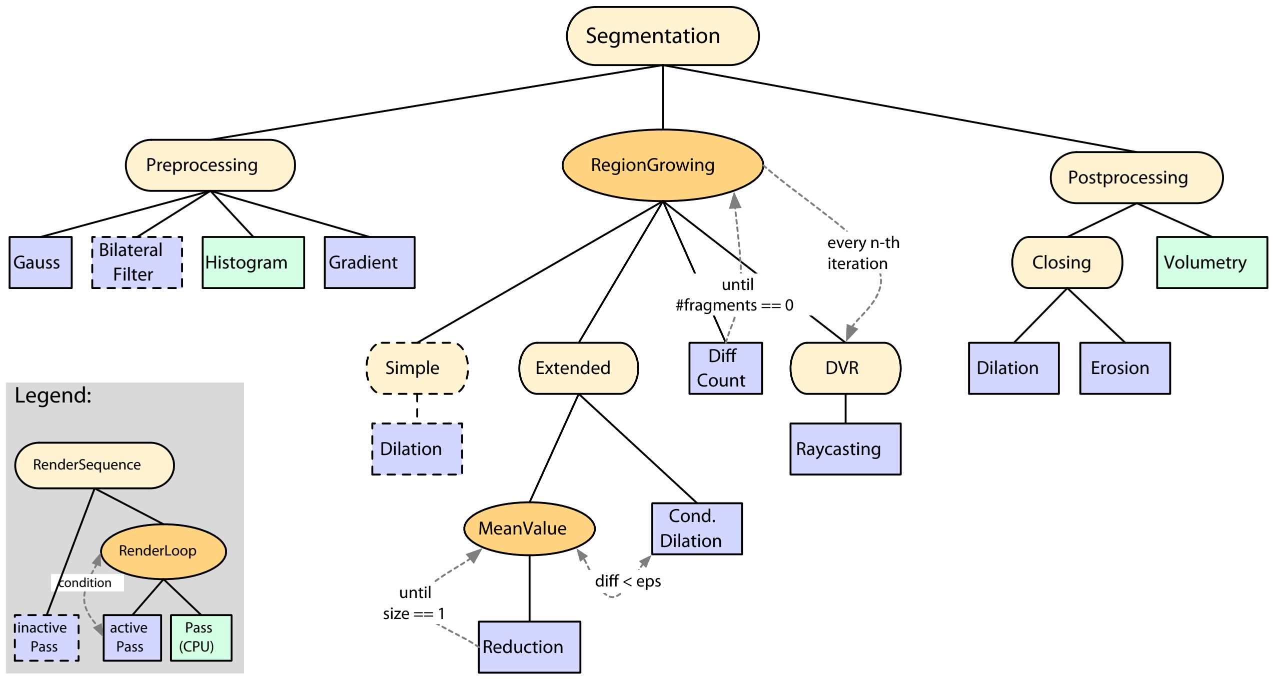

Fig. 2 Example workflow for a volume segmentation procedure using our representation and depicted as tree. Note that some steps have been integrated as CPU implementations to show the interchangeability of the modules. In addition, a direct visualization component (DVR) is available, with negligible overhead for GPU computations.

and the `RenderLoopCondition` can be set accordingly. Also, a visualization sequence is executed every n-th iteration, which comes at almost no performance overhead for GPU computations as no image/volume data has to be transferred. Finally, some postprocessing is applied to the segmentation data: a typical morphological operation, followed by counting the number of voxels segmented in our case.

As described in the preceding section, this approach also handles the parameters involved in such a sequence. In our example, global parameters are the texture size (for neighbor access) or statistical values of the data itself. Local parameters, however, are specific to the individual sequences and/or passes: a threshold for the region growing, a structuring element for the morphological operations, etc. As mentioned before, different parts of the tree can be reused, either separately (e.g. the DVR sequence) or in other sequences (e.g., preprocessing).

5 Results

Performance issues are often mentioned as a downside of design patterns and other object-oriented programming techniques that involve indirections or make use of virtual methods. To measure the overhead introduced by our approach, we have extracted all OpenGL commands from a region growing sequence similar to the one discussed in section 4. Then we compared the performance of consecutively executing these OpenGL commands with executing the original sequence within our framework. This leads to a run time performance overhead of ca. 10%: On average, the whole sequence takes 2.02 seconds (pure OpenGL) and 2.23 seconds (Cascada), respectively.

Although additional run time costs of that order cannot be ignored, it should be taken into account that using a framework to integrate shader programming with the host application greatly simplifies the implementation, especially of more complex algorithms. Thus, development time can be reduced remarkably, outweighing the performance loss by far. It should also be noted that the extracted code can be handled differently by compilers than the complete application code. In addition, our implementation has not been optimized yet allowing further improvements and reducing the gap.

6 Conclusion

We have proposed an approach to represent computations on programmable graphics hardware hierarchically to allow for a problem-oriented description of algorithms. In this context, we have introduced object-oriented concepts to GPU programming, while preserving shader programs as flexible and extensible component. Finally, we have discussed the possibilities and results of our approach by means of an example from volume segmentation.

For the future we would like to further investigate the potential of generalizing shader programming. Therefore, we plan to realize more (image/volume processing) algorithms using Cascada and extend its capabilities with respect to graphics-oriented data types, performance issues, etc. We also aim at assessing the possible benefit and overall limitations of such implementations. Related to that is an extended comparison of our approach with systems like Nvidia's CUDA and the adequate integration of CPU functionality.

Acknowledgements We would like thank MeVis Research in Bremen, Germany, for their collaboration and especially Alexander Köhn, Felix Ritter and Florian Link for support and discussion. Also many thanks to Stefan Müller and Dirk Reiners for their invaluable advice and comments on our work, as well as several students contributing to the development of Cascada.

References

1. Abram, G.D., Whitted, T.: Building Block Shaders. In: SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pp. 283–288. ACM, New York, NY, USA (1990). DOI http://doi.acm.org/10.1145/97879.97910
2. Cook, R.L.: Shade Trees. In: SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 223–231. ACM, New York, NY, USA (1984). DOI http://doi.acm.org/10.1145/800031.808602
3. Eissele, M., Weiskopf, D., Ertl, T.: The G2-Buffer Framework. In: Proceedings of SimVis, pp. 287–298 (2004)
4. Fernando, R., Kilgard, M.J.: The Cg Tutorial – The Definite Guide to Programmable Real-Time Graphics. Addison-Wesley Professional (2003)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)
6. Kuck, R.: Object-Oriented Shader Design. In: J.S. P. Cignoni (ed.) Proceedings of Eurographics 2007, pp. 65–68. Eurographics, The Eurographics Association (2007)
7. McCool, M.D., Qin, Z., Popa, T.S.: Shader Metaprogramming. In: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 57–68. Eurographics Association, Aire-la-Ville, Switzerland (2002)
8. McGuire, M., Stathis, G., Pfister, H., Krishnamurthi, S.: Abstract Shade Trees. In: I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, pp. 79–86. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1111411.1111425
9. Trapp, M., Döllner, J.: Automated Combination of Real-Time Shader Programs. In: J.S. P. Cignoni (ed.) Proceedings of Eurographics 2007, pp. 53–56. Eurographics, The Eurographics Association (2007)