HIGH-LEVEL GPU PROGRAMMING

Domain-specific optimization and inference

Calle Lejdfors

Doctoral dissertation, 2008
Department of Computer Science
Lund University

ISBN 978-91-628-7511-4
ISSN 1404-1219
Dissertation 29, 2008
LU-CS-DISS:2008-2

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: calle.lejdfors@cs.lth.se
WWW: http://www.cs.lth.se/home/Calle_Lejdfors/

Typeset using LaTeX2ε.
Printed in Sweden by Tryckeriet i E-huset, Lund, 2008.

Abstract

When writing computer software one is often forced to balance the need for high run-time performance with high programmer productivity. By using a high-level language it is often possible to cut development times, but this typically comes at the cost of reduced run-time performance. Using a lower-level language, programs can be made very efficient but at the cost of increased development time.

Real-time computer graphics is an area where there are very high demands on both performance and visual quality. Typically, large portions of such applications are written in lower-level languages and also rely on dedicated hardware, in the form of programmable *graphics processing units* (GPUs), for handling computationally demanding rendering algorithms. These GPUs are parallel stream processors, specialized towards computer graphics, that have computational performance more than a magnitude higher than corresponding CPUs. This has revolutionized computer graphics and also led to GPUs being used to solve more general numerical problems, such as fluid and physics simulation, protein folding, image processing, and databases. Unfortunately, the highly specialized nature of GPUs has also made them difficult to program.

In this dissertation we show that GPUs can be programmed at a higher level, while maintaining performance, compared to current lower-level languages. By constructing a *domain-specific language* (DSL), which provides appropriate domain-specific abstractions and user-annotations, it is possible to write programs in a more abstract and modular manner. Using knowledge of the domain it is possible for the DSL compiler to generate very efficient code. We show that, by experiment, the performance of our DSLs is equal to that of GPU programs written by hand using current low-level languages. Also, control over the trade-offs between visual quality and performance is retained.

In the papers included in this dissertation, we present domain-specific languages targeted at numerical processing and computer graphics, respectively. These DSL have been implemented as *embedded languages* in Python, a dynamic programming language that provides a rich set of high-level features. In this dissertation we show how these features can be used to facilitate the construction of embedded languages.

Preface

This dissertation is based on the following papers, which will be referred to in the text by their Roman numerals. The papers are appended at the end of the dissertation.

I Calle Lejdfors and Lennart Ohlsson, PyGPU: A high-level language for high-speed image processing, In *Proceedings of IJCSIS '07*, pages 66–81, 2007.

II Calle Lejdfors and Lennart Ohlsson, Implementing an embedded GPU language by combining translation and generation, In *Proceedings of SAC '06*, pages 1610–1614, 2006.

III Calle Lejdfors and Lennart Ohlsson, Unified GPU programming, *Technical report, Lund University, LU-CS-TR:2008-244*, 2008. Extended version of a paper accepted for publication in *Proceedings of CGV '08*.

IV Calle Lejdfors and Lennart Ohlsson, Towards modular programming of the GPU pipeline, *Manuscript in preparation*, 2008.

V Calle Lejdfors and Lennart Ohlsson, Space-free shader programming: Automatic space inference and optimization for real-time shaders, Accepted for publication in *Proceedings of TPCG '08*, 2008.

VI Calle Lejdfors and Lennart Ohlsson, Separating shading from rendering in real-time graphics applications, *Manuscript in preparation*, 2008.

In addition, the following papers are written, or contributed to, by the author:

– Henrik Malm, Magnus Oskarsson, Eric Warrant, Petrik Clarberg, Jon Hasselgren, Calle Lejdfors, Adaptive enhancement and noise reduction in very low light-level video, In *Proceedings of ICCV '07*, pages 1–8, 2007.

– Calle Lejdfors and Lennart Ohlsson, PyFX: A framework for real-time graphics effects, *Technical report, Lund University, LU-CS-TR:2005-233*, 2005.

– Calle Lejdfors and Lennart Ohlsson, PyFX – An active effect framework, In *SIGRAD 2004 Conference proceedings*, pages 17–24, 2004.

Acknowledgments

The work presented in this dissertation was carried out within the Computer graphics group at the Department of Computer Science, Lund University. I would like to thank my supervisors, Assistant Professor Lennart Ohlsson, Assistant Professor Görel Hedin, and Professor Boris Magnusson for allowing me to freely pursue research into those areas that I found interesting. In particular, I wish to thank Lennart Ohlsson for many rewarding discussions on software design in general, as well as for helpful suggestions when preparing this manuscript and the articles contained herein.

Also, I would like to thank the rest of the graphics group (Tomas, Jon, Petrik, Jacob, and Jim) along with the rest of the department for creating an inspiring and relaxing working climate.

Finally, I am deeply grateful to Ingela Carlgren, the light of my life, for her unwavering love and support. Also, to my family and all my friends: thank you for reminding me about what is important in life.

Contents

Introduction

When writing computer software, a balance must be struck between two complementary goals: run-time performance and programmer productivity. High-level languages increases productivity by providing abstractions and features that simplify programming. However, often these features comes with an abstraction penalty in the form of decreased run-time performance. Lower-level languages offer great performance and enable fine-grained control over the target hardware. But this control comes at the expense of an increased burden on the programmer.

Performance-sensitive applications are typically written in lower-level languages to obtain maximal performance. For example, real-time computer graphics applications are typically implemented in C/C++, or even assembler, to answer the ever increasing demands on performance, visual quality, and interactivity. Such applications generally also rely on dedicated programmable *graphics processing units* (GPUs) for offloading computationally demanding rendering algorithms from the main CPU.

These GPUs are highly specialized, parallel stream processors that have a computational performance more than a magnitude higher than that of CPUs. For programming these processors a number of specialized C-like languages are available (Cg [44], HLSL [25], and GLSL [57]). This combination of flexible programmability and high performance has led to a revolution in computer graphics, allowing many realistic rendering algorithms to be executed in real-time [10, 20, 54, 49, 21]. Also, GPUs have been adapted to be used as co-processors for solving more general numerical problems [39, 42, 9].

However, due to their highly specialized nature, the programming model of GPUs is very restricted. There is no heap or stack, no flexible addressing, and limits on where data may be written. Using current languages these limitations must be explicitly addressed in the program source code. This makes GPU programs difficult to write, modify, and manage, a problem in modern graphics applications [48, 3].

In this dissertation we show that by constructing *domain-specific languages* (DSLs) it is possible to program the GPU at a higher level, while maintaining performance compared to current GPU languages. A DSL provides a set of abstractions and user-annotations targeted at a specific problem domain. Together with knowledge of the

target domain, a compiler can use these annotations to perform domain-specific optimization and analysis to ensure both performance and correctness.

In the papers included in this dissertation, we present a number of DSLs targeted at general purpose computations on the GPU (Papers I and II) and real-time graphics (Papers III-VI), respectively. These languages enable GPU programs to be constructed in a more abstract and modular fashion. We show, by experiment, that the performance is equal to that of programs hand-written in existing GPU languages. Also, in the case of rendering, we demonstrate that control over the trade-offs between performance and visual quality, can be maintained compared to lower-level approaches.

The presented prototype DSLs are all implemented as *embedded languages* [17]. Language embedding enables quick experimentation with different language design choices and compiler implementation techniques. For this we have used the dynamic programming language Python [43]. Dynamic languages typically provide a rich set of high-level programming constructs together with extensive support for *introspection* (i.e. the ability of a program to observe and manipulate its own execution) making them well suited for implementing embedded languages. In this dissertation, we show how these features can be used to enable straightforward construction of embedded domain-specific languages.

The rest of this introduction is structured as follows: In Section 1 we describe domain-specific languages in more detail. In Section 2 we give a brief introduction to dynamic programming languages. Next (in Section 3) we introduce modern graphics programming using the GPU. In Section 4 we summarize the contribution made in this dissertation and then, in Section 5, we discuss our results and give some ideas for future work.

1 Domain-specific languages

The earliest uses of domain-specific languages were in compiler-related tools such as parser and scanner generators (Lex [40], YACC [34], and Zephyr [70]) and in compiler construction (JastAdd [27], UUAG [4]). Other examples of DSLs are in database query languages (SQL [11], dBase [35]), symbolic algebra (Maple [12]), and parallel programming (Orca [6]). Also, languages such as LISP [61] and FORTRAN [5] started out as domain-specific languages, targeted at artificial intelligence and numerical code, respectively, but gradually evolved into more general languages. In relation to this, we see that it is not obvious what constitutes a domain-specific language and what does not. Van Deursen et al. [67] proposes the following definition:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

The use of the term *problem domain* here is rather vague. For instance, a language such as MATLAB [45] can be said to be a DSL targeted at array programming. The key distinction is that DSLs are usually *small*, only providing a limited set of abstractions and operations, highly targeted at the problem domain. Neither LISP, FORTRAN, nor Matlab are small languages; their expressive power is not restricted to their respective initial domains. Also, DSLs generally contain assumptions, and perform optimizations, that are not valid in the general case.

Using a language specialized for the problem at hand comes with a number of benefits. DSLs allow problems to be expressed using the idioms inherent to the problem domain. This enables non-programmer domain experts to understand, modify, and sometimes even develop DSL programs. A DSL can be very concise, expressing in a few lines what would require several hundred lines in a general language, and this increases productivity, reliability, and maintainability. Furthermore, since the DSL compiler has access to domain-specific knowledge, it can perform optimization and validation at the domain level.

The advantages of domain-specific languages come at a rather large initial cost. Designing and implementing a new language from the ground up is a costly venture. There is a high likelihood that the initial designs will not be good enough. The language will evolve and change with user demand for more or different functionality. Following the argument of Hudak [29], this will lead to a higher initial cost compared to conventional methods. However, when viewed over the whole life-cycle of a software project, the total implementation cost is reduced.

1.1 Embedded domain-specific languages

One approach to lowering the initial cost of implementing a domain-specific language is reusing the infrastructure of an existing language. An *embedded* DSL, also known as a domain-specific embedded language (DSEL) [30], can be created in terms of an existing *host* language. By doing so, a large part of the existing functionality of the host language can be reused. This includes aspects such as syntax, type-checking, evaluation model, etc. Consequently, using an embedded approach enables the implementor to focus on the domain-specific idioms and abstractions, leaving more general details to the host language.

Implementation strategies for embedded languages

When implementing an embedded DSL, there is one important factor to consider: whether the language should target the same platform as the host language, or a different one. If the target coincides with that of the host language then the most common strategies are using either *macros* or *domain-specific libraries*. The macro facilities of LISP and Scheme [1] have long been used to create constructs having the "look-and-feel" of built-in primitives. Macros have been used to implement embedded languages

and language extensions such as the Common LISP Object System [38] (for object-oriented programming in LISP) and Verischemelog [33] (an embedded language for digital hardware design in Scheme).

When using a domain-specific library approach, the operator and function overloading facilities of the host language are used to construct the embedded language. Functional languages, such as Haskell [36], that provide extensive overloading and operator redefinition facilities, are popular with this method. To name a few examples of languages embedded in Haskell, and their respective domains: FRAN [18] and Yampa [31] (reactive animations), Haskore [32] (music), and Lava [7] (hardware description and verification). The industry standard C++ [62] has also been used to implement embedded languages; for example TBAG [19], for interactive animations, and Spirit [60], a template meta-programming language [15] for constructing parsers.

When the target is *not* the same platform as that of the host language then the above approaches can not be used. Instead a *generative* approach is typically employed, where programs in the host language are executed to generate programs for the DSL target platform. This is the approach taken by the Haskell-embedded languages Pan [17], for image manipulation and synthesis, and Vertigo [16], for programming the vertex processing functionality of the GPU . Both these languages use the overloading facilities of Haskell to construct tree-representations that are compiled to their respective target platforms: Photoshop plug-ins and GPU vertex programs. In C++, Sh [46] is a language for programming GPUs. It uses a technique where functions are executed and have their execution recorded. This recording is then used as input to a compiler responsible for generating GPU assembler code. A similar method is used by Accelerator [64], an embedded language for data-parallel computations on the GPU, available in C# [28].

The problem with using a generative approach is that much information of the program is lost. For example, when embedding in a procedural language, since the host level code is executed to generate the intermediate representation, host language constructs such as conditionals and loops will not appear in the recorded stream of operations. Hence, they can not be translated. Instead, a host language conditional is equivalent to conditional compilation in the embedded language.

Recently, Microsoft's C# and Visual Basic have introduced support for constructing an *expression tree* representation of a piece of code at compile-time. These trees can be accessed and examined by the program at run-time and be used to implemented a DSEL compiler, for instance. Expression trees, combined with a specialized query syntax, is the basis of LINQ [66] a method for querying data structures and databases integrated into C# and Visual Basic.

2 Dynamic languages

Dynamic languages, such as Python, Ruby [65], LISP, and Smalltalk [23], are languages designed to increase programmer efficiency. Dynamic languages enable faster development cycles by allowing parts of a program to be modified at run-time. Functions may be introduced, removed, or changed, classes added, class inheritance modified, and modules can be created or removed. This allows a programmer to quickly test a new feature, or a new piece of code. Furthermore, modern dynamic languages, such as Python and Ruby, provide syntax and high-level programming models that are easy to learn, resulting in higher productivity.

Since every part of a program can be changed at run-time, dynamic languages are generally *interpreted*, looking up symbols and code at run-time. Consequently, dynamic languages also tend to be *dynamically typed*, i.e. they perform type checking at run-time. Dynamic typing has the advantage of making code clearer and shorter, since the syntactic clutter of explicit type information is avoided. Furthermore, since functions are written without type information, code can be reused on many types of objects. This is achieved without introducing a common base-class or interface, giving very fine-grained requirements of a function; a function requires only those operations it uses, no more, no less. This leads to an alternate name for dynamic typing; *duck typing*, inspired by the metaphor "if it looks like a duck and quacks like a duck, it must be a duck" [68]. These features together with the fact that dynamic languages tend to be higher-level than traditional static languages, lead to reduced development time and, consequently, increased programmer productivity.

The downside of using dynamic languages is performance. Since symbol lookup and type checking is performed at run-time, there is a higher run-time overhead. Furthermore, since the types of a function's arguments are not known statically, generating efficient native CPU code for dynamic languages is a very difficult problem [56, 58]. Also, the lack of static types makes compile-time checking and verification hard.

Meta-programming facilities

The fact that type checking and variable lookup occurs at run-time implies that the machinery for doing this is, at least partially, available to the programmer. The meta-level ability of a running program to observe itself and its own execution is usually referred to as *introspection*. Introspection is an integral part of LISP and Scheme and it is supported by most other dynamic languages. Some other languages, notably Java [37], C#, and the C++-extension OpenC++ [14], also support introspection. In addition, dynamic languages tend to also support *intercession*, allowing parts of a running program to be modified. Collectively, introspection and intercession are called *reflection*.

The most common use of introspection is in the development of programming

tools such as class browsers or debuggers. More interestingly, reflective languages facilitates the adaption of the language to the application domain. Using reflection it is possible to meta-programmatically extend an object-oriented language with additional functionality such as multi-method dispatch [69], and aspect-oriented programming features [63]. Also, as shown in this dissertation, introspection can be used to greatly facilitate the implementation of embedded languages by making more structural information available to the compiler of the embedded language.

3 Graphics programming

Real-time three-dimensional computer graphics is usually done using a dedicated graphics card, responsible for drawing, or *rasterizing*, polygons to the screen. Modern graphics cards also have a programmable graphics processing unit (GPU) that enables parts of the rasterization process to be programmatically redefined. For a thorough introduction to graphics card programming, see "The Cg Tutorial" by Fernando and Kilgard [55] or the "OpenGL Programming Guide" by Shreiner et al. [59].

3.1 3D graphics cards

3D graphics cards implement a graphics *pipeline* (see Figure 1) consisting of multiple stages where polygons are transformed, clipped, textured, and finally rasterized to the screen. Typically, each stage consists of multiple parts where each part can be turned either on or off. Also, the different parts can be controlled to some limited degree. For example, the depth testing part of the pixel processing stage can be turned on or off, and be instructed to use different functions for determining whether a pixel is visible or not.

By controlling the different stages of the pipeline, an application can achieve a wide range of different visual effects. For example, texturing, transparency, antialiasing, and line stippling can all be controlled by setting one or more states to the appropriate values. For controlling these pipeline stages, graphics APIs, such as OpenGL [59] and DirectX [8], are supported by all major graphics card vendors.

3.2 The graphics processing unit

In the late 1990s and early 2000s, the increasing demands of real-time graphics applications, typically games, led to the introduction of a large number of specialized hardware extensions. These extensions each provide a specific feature required for increased visual acuity over that provided by the original fixed-function pipeline. Examples of such extensions are per-pixel lighting, bump-mapping, and vertex weighting used for skinning. As an interesting comparison, today there are over 300 different OpenGL extensions. A more general method for handling extensions was needed.

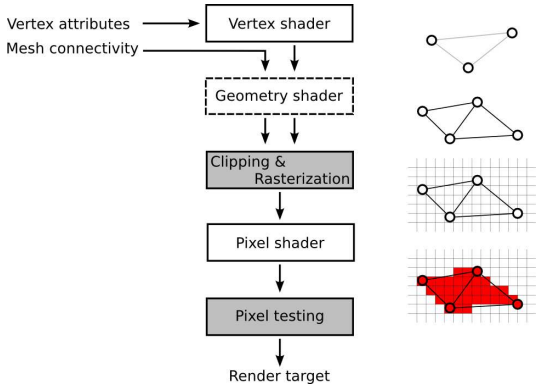


Figure 1: A schematic view of the rasterization pipeline of a modern GPU. White boxes represent programmable stages and grey boxes indicate fixed functionality. The stippled geometry shader stage is optional and need only be used if dynamic geometry generation or re-tessellation on the GPU is required. The example triangles on the right indicate how a single primitive is transformed by the different stages.

This led to the introduction of programmable graphics hardware, commonly called the graphics processing unit (GPU), in 2001 [41]. The first GPU generations allowed the vertex and pixel processing stages of the graphics pipeline to be programmatically redefined. This programmability simultaneously allowed most of the extensions previously introduced to be emulated, and expanded the boundaries of the kind of real-time visual effects that were possible.

Today the scope of what kind of effects are possible has been further expanded with the introduction of the *geometry processing unit* [8] that enables primitives (points, lines, triangles, etc.) to be generated and manipulated as part of the GPU pipeline. This makes it possible to dynamically re-tessellate meshes or procedurally construct geometry on the GPU, for example.

Programs running on the GPU are called *shader programs*, originally referring to the act of coloring, or *shading*, a pixel. The *vertex shader* processes the vertices of the input mesh one by one, and computes data that is interpolated across each primitive. The interpolated data is used as input to the *pixel shader*[1], which computes the final pixel color. If used, the *geometry shader* accepts as input single primitives constructed from output vertices of the vertex shader. The output is zero or more new primitives, which need not be of the same type as the input. The primitives generated by the geometry shader are rasterized and passed on to the pixel shader.

[1] In OpenGL this is known as a *fragment shader* since it processes a complete data fragment (including, for instance, color, texture coordinates, and depth information). The distinction is not important in the context of this work and the term pixel shader will be used from here on.

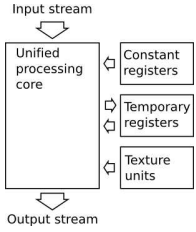


Figure 2: A schematic view of a unified GPU processing core. Usually these cores are implemented as stream processors where the final memory location of the output can not be changed or manipulated.

3.3 GPU programming

Modern GPUs are aggressively pipelined processors with raw floating-point performance more than an order of magnitude higher than corresponding CPUs [39]. Also, the increase in performance between successive generations is higher than that of CPUs. Together this makes the GPU a computationally powerful platform, both for graphics applications and more general purpose ones.

The first generations used separate vertex and pixel processors for executing the vertex and pixel shader programs, respectively. Recent GPUs generally use a *unified shader model* where each programmable pipeline stage is executed on the same physical processing cores. Sharing cores between the different pipeline stages enables the GPU to do load-balancing to match the balance of computations for a particular shader. For example, for effects that require a high amount of pixel shader computations, correspondingly more cores can be assigned to the pixel processing stage. For an overview of a unified processing core of a modern GPU, see Figure 2.

The high performance of the GPU comes from it being both highly parallel and aggressively pipelined. This imposes certain restrictions on the level of programmability of the GPU. When using shaders, there is no support for globally writable random-access memory. Memory may be read only from 1, 2, or 3-dimensional texture maps. There is no stack or heap, and no support for pointers. All computations are performed using a set of registers dedicated to input, output, or temporary storage.

The lack of writable memory put severe restrictions on the programmability of the GPU. For instance, it is not possible to use objects or even lists, since they both rely on dynamic memory allocation. Furthermore, a pixel program may only write information to a single pixel on screen. The position of this pixel is determined earlier in a fixed-function part of the pipeline and can not be changed (if desired, the pixel may be discarded however). This lack of *scattered* writes has implications primarily for using the GPU for more general computational tasks.

3.4 GPU programming languages

The first generations of GPUs were programmed using assembler languages specific to each vendor. Now, NVIDIA's Cg, Microsoft's HLSL, and GLSL by the OpenGL ARB, provide C-like languages for programming the GPU. These enable shader programs to be written at a higher level than previously possible. However, to clarify, these languages are not domain-specific languages. They are general purpose languages that target the GPU. The make only minimal assumptions on the structure of the GPU and do not perform any domain-specific optimization or analysis (such as moving code between different pipeline stages, tracking/inferring the coordinate frame of a vector quantity, or automatically normalizing interpolated unit vector data). One reason for this is to maintain a high degree of programmer control. Also, it provides a clear path by which future features can be supported [44].

Also, as mentioned above, Sh and Vertigo provide GPU programming languages embedded in C++ and Haskell, respectively.

3.5 General purpose GPU computations

The programmability and high performance of GPUs have spawned great interest in using them for computationally intensive, non-graphics tasks. Algorithms, such as image processing [53], fluid simulation [26], behavioral models [13], audio processing [71], database operations [24], and numerical solvers [22], have all been implemented to run on GPUs. Using a GPU for non-graphics tasks, such as simulation or numerical computations, is collectively known as general purpose GPU computing, abbreviated *GPGPU*.

The main advantage of using the GPU is performance. By using the pipelined, highly parallel architecture it is possible to achieve raw numerical floating-point performance an order of magnitude greater than current CPU generations. Moreover, the speed increase with each new GPU generation is greater than the corresponding increase in CPU processing power. This makes the GPU a very attractive platform for computationally intensive algorithms.

Unfortunately, as explained in Section 3.3, the programming model of the GPU is quite restricted. Also, the GPU is optimized to work with graphics primitives such as vertices, triangles, and textures. Until recently, porting an algorithm to run on the GPU entailed expressing it in terms of these primitives. This has changed with the introduction of NVIDIA's CUDA [50] and AMD's Close-to-Metal (CTM) [52] languages, that provide a programming model that is closer to the actual hardware of the GPU. These languages lift some of the restrictions inherent in expressing an algorithm as operations on graphics primitive. For example, it is possible to write to, and read from, arbitrary memory locations on the GPU, assuming synchronization of overlapping parallel writes is handled.

However, regardless of the implementation technique used, to use the GPU effi-

ciently, the programmer must be aware of the minutiae of how the GPU is structured: number of stream processors per block (a group of processors with fast shared memory and synchronization facilities), total number of blocks, performance of using different types of memory, synchronization, and so on. This implies that, in order to use the GPU as a numerical co-processor, an implementor must have in-depth knowledge of both the target domain, e.g., numerical algorithms or audio processing, and advanced knowledge of GPU programming. These limitations have severely limited the spread and uptake of GPGPU techniques.

In addition to CUDA and CTM, other projects have previously been proposed for facilitating the use of the GPU in numerical processing. BrookGPU [9] is a compiler for writing numerical GPU algorithms. The Brook language is an extension of C that incorporates the primitives *streams*, representing data, and *kernels*, representing operations on streams, for expressing data parallel computations. Sh [47] provide a similar C++ template-based abstraction primarily targeted at implementing GPGPU algorithms. Similarly, as mentioned Accelerator provides an array interface to programming the GPU from C#.

4 Contributions

This dissertation focus on techniques and methods for high-level programming of the graphics processing unit. To achieve this we use domain-specific languages embedded in the dynamic language Python. The use of language embedding enables us to implement prototype DSLs quickly, allowing us to test different language design choices and implementation strategies.

We have focused primarily on three aspects of high-level programming:

- *Abstraction* Abstractions that simultaneously provide a practical programming model and are sufficiently close to the target hardware to enable translation to efficient low-level code.
- *Modularization* By building programs as combinations of well-defined modules, that can be maintained and developed separately, it is possible to increase code reuse and reduce development time.
- *Performance* The primary uses of GPUs are in very performance-sensitive areas such as real-time rendering and numerical simulation. Hence, performance is a very important consideration.

In particular, our goal is to support high levels of abstraction and modularization while maintaining performance equal to that of hand-written GPU programs.

In Papers III to VI, we have tried to keep the syntax and semantic models of our DSLs close to those of existing GPU languages [25, 44, 57]. Our wish have been

to minimize the amount of user-annotations that are required and instead focus on maximizing the amount of information that can be automatically inferred by the compiler. For example, in Papers III, IV, and VI, we only require that the programmer distinguish between unit vectors and directions. This is necessary for correct interpolation of such data across the primitives. In Paper V we extend this classification by also mandating that the space in which a vector is expressed must be specified when that information can not be automatically inferred. These small extensions are sufficient to enable a compiler to perform significantly more domain-specific analysis and optimization than is possible using only structural data types.

Next, our contributions in each of the above areas are presented in separate subsections. The author of this dissertation is the primary author of all the included papers. Also, he is responsible for the implementation of all the presented DSLs and compilation techniques, as well as the vast majority of the examples.

4.1 Abstraction

In Papers I and II we present PyGPU, a DSL embedded in Python that enables high-level programming of image processing algorithms on the GPU. We present a novel functional image abstraction that, by construction, encapsulates the most important restriction of the GPU; the lack of scattered writes. In Paper I we present a number of examples of using this abstraction to implement several non-trivial algorithms to take advantage of the performance of the GPU. The compilation strategy used by PyGPU is described in Paper II. It uses the introspection support of Python to make more structural information available to the compiler. This enables us to have the simpler implementation of a generative approach while still being able to translate elements such as loops and conditionals (see Section 1.1).

In Papers III and IV, we introduce the unified programming model where the programmer need not be aware of the separate GPUs stages. Shaders are written as single programs that the compiler analyzes, splits, and finally maps to the appropriate pipeline stages. Compared to previous methods, our approach does not require any user-annotations about the placement of operations. Instead the compiler uses knowledge of the domain to ensure that the generated GPU programs are correct. Also, our approach extends previous methods by supporting loops, conditionals, vertex texture fetches, etc., available on modern GPUs.

Paper V introduces a novel optimization and analysis strategy that enables shaders to be written without explicit space-transformations. Instead, our compiler uses information about the application to infer the best, correct choice of space (i.e. the same choice that an experienced programmer would make). This simplifies programming and reduces application dependencies. Also, it represents a optimization strategy not previously considered.

4.2 Modularization

The PyGPU language introduced in Papers I and II enables straightforward mixing of GPU and CPU computations by making PyGPU computations map to ordinary Python functions. This enables performance-critical part of an application to be rewritten using PyGPU with minimal impact on the rest of the application.

In Paper IV we present a programming model that enables the GPU pipeline to be programmed in a more modular fashion, allowing geometry generation to be specified separately from the actual surface shading. This enables the geometric and shading parts of an effect to be developed as separate, composable modules that can be combined to yield new effects. To our knowledge, this is the first real-time shading language that supports modular programming in this way. This paper also introduces a novel analysis algorithm for factorizing shaders across all stages (vertex, pixel, and geometry) of a modern GPU.

In Paper VI we present a framework capable of automatically rewriting an existing shader to use any combination of the shader-based rendering techniques: instanced rendering, deferred shading, and vertex stream-out caching. This reduces code duplication by eliminating the need to explicitly combine rendering with shading code, and increases code reuse.

4.3 Performance

The overarching goal of all research presented in this dissertation has been to keep the introduced performance penalty very low, preferably non-existent. We succeed in this. In Papers III, IV, and VI the performance of our approaches is equal to that of hand-written shader programs implemented in current lower-level GPU languages. The inference method introduced in Paper V does have a small performance hit, but only on the order of a few percent (see Table 2, page 95). In PyGPU (Papers I and II) we achieve very good performance numbers even though the compiler does not yet implement any advanced optimization techniques.

Furthermore, we address a common objection to high-level programming models and languages: loss of control. In high performance applications, in particular, this loss of control is often coupled with a loss of performance. In GPU programming this is aggravated by the added complexity of being able to perform operations at different stages of the rendering pipeline, choices that have a large impact on both the performance and visual quality of the resulting shader.

In Papers III and IV we provide a high-level programming model where this level of control is retained. Instead of relying on the programmer to spread computations across the pipeline stages manually, an initial placement is inferred by our compiler. This placement can then be controlled through an external interface such as an artist's GUI. This enables the balance between performance and visual quality to be controlled to the same degree as when writing shader programs by hand, but without

requiring any source code changes. Hence control is retained, and development time is reduced. This kind of external control mechanism has not been considered previously.

5 Conclusions and future work

We have shown that it is possible to have a high-level programming model of the GPU while maintaining performance compared to using current GPU languages. By using domain-specific analysis and optimization it is possible for a compiler to generate very efficient code from such higher-level programs. Also, we show that control over the trade-offs between shading performance and visual quality can be maintained. Moreover, this control is achieved without requiring any rewriting of the shader source code. As demonstrated, this control can be integrated in an artist's tool thereby allowing shader programs to be manipulated as part of the asset authoring process.

In all the papers presented in this dissertation, we have used domain-specific embedded languages implemented in the dynamic language Python. Thus we have shown, by several examples, that it is possible to implement very high performance DSLs even in a dynamic host language. In particular, the introspection ability of Python have significantly simplified the implementation and testing of different language features and design choices.

For future research, a number of areas are of interest. For example, combining the space analysis and inference from Paper V with the "building block" approaches for constructing shaders [2, 47, 48]. This could potentially lead to higher performance (by letting the compiler choose an appropriate block implementation for the optimal space) and more general code (by writing blocks that work for an arbitrary choice of space). Similarly, combining the modular approach of Paper IV with the block-based methods is a natural extension.

Also, investigating other code generation targets for the PyGPU language presented in Papers I and II is an interesting research venue. For example, translating to CUDA or CTM can potentially give significant performance gains compared to the currently used technique of mapping programs to operations on pixels and textures.

Mapping other types of programming paradigms to the GPU would also be a natural evolution of the techniques and results presented in this dissertation. For example, automatic translation of array programming (such as provided by the NumPy [51] library or MATLAB [45], for example) could potentially lead to a significant performance increase compared to direct interpretation or translation to CPU code.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2nd edition, 1996.

[2] Gregory D. Abram and Turner Whitted. Building block shaders. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 283–288, 1990.

[3] Johan Andersson and Natalya Tatarchuk. Frostbite rendering architecture and real-time procedural shading & texturing techniques. AMD Sponsored Session. GDC 2007, March 2007.

[4] Arthur Baars, Doaitse Sweirstra, and Andres Löh. *UU AG System User Manual*. Department of Computer Science, Utrecht University, September 2003.

[5] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference, February 26–28, 1957, Los Angeles, CA, USA*, pages 188–198. pub-IRE, 1957.

[6] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, 1992.

[7] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, 1998.

[8] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[9] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[10] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *Graphics hardware*, pages 51–59. Eurographics Association, 2003.

[11] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM Press, 1974.

[12] Bruce Char, Keith Geddes, and Gaston Gonnet. The Maple symbolic computation system. *SIGSAM Bull.*, 17(3-4):31–42, 1983.

[13] Rosario De Chiara, Ugo Erra, Vittorio Scarano, and Maurizio Tatafiore. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. In Bernd Girod, Marcus A. Magnor, and Hans-Peter Seidel, editors, *VMV*, pages 233–240. Aka GmbH, 2004.

[14] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, 1995.

[15] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[16] Conal Elliott. Programming graphics processors functionally. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 45–56, 2004.

[17] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27. Springer-Verlag, 2000.

[18] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[19] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. Tbag: a high level framework for interactive, animated 3d graphics applications. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 21, pages 421–434, 1994.

[20] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[21] Alfred R. Fuller, Hari Krishnan, Karim Mahrous, Bernd Hamann, and Kenneth I. Joy. Real-time procedural volumetric fire. In *Symposium on Interactive 3D graphics and games*, pages 175–180, 2007.

[22] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE SC'05 Conference*, 2005.

[23] Adele Goldberg and David Robson. *Smalltalk 80: The Language*. Addison-Wesley Professional, 1989.

[24] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, 2004.

[25] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[26] Mark J. Harris. *GPU Gems*, chapter 38, pages 637–665. Addison-Wesley Professional, March 2004.

[27] Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[28] Anders Hejlsberg. *The C# programming language*. Addison-Wesley Pub Co, 1 edition, 2003.

[29] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134. IEEE Computer Society, 1998.

[30] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4):196, 1996.

[31] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[32] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[33] James Jennings and Eric Beuscher. Verischemelog: Verilog embedded in scheme. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 123–134, 1999.

[34] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[35] Edward Jones. *The dBase language reference*. Osborne/McGraw-Hill, 1990.

[36] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. ISBN: 0521826144.

[37] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *Java™ Language Specification*. Addison-Wesley Pub Co, 2nd edition, 2000.

[38] Gregor Kiczales, J. Michael Ashley, Jr. Luis H. Rodriguez, Amin Vahdat, and Daniel G. Bobrow. *Metaobject protocols: why we want them and what else they can do*, pages 101–118. MIT Press, 1993.

[39] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[40] Michael E. Lesk and Eric Schmidt. lex: A lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 288–400. Holt, Rinehart, and Winston, 1979.

[41] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 149–158, 2001.

[42] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3d fluid simulation on GPU with complex obstacles. In *Proceedings of Pacific Graphics 2004*, pages 247–256, October 2004.

[43] Mark Lutz. *Programming Python*. O'Reilly, 3 edition, 2006.

[44] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[45] Matlab. http://www.mathworks.com/.

[46] Michael McCool, Zheng Qin, and Tiberiu Popa. Shader metaprogramming. In Thomas Ertl, Wolfgang Heidrich, and Michael Doggett, editors, *Graphics Hardware*, pages 1–12, 2002.

[47] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[48] Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract shade trees. In *I3D '06*, pages 79–86, New York, NY, USA, 2006. ACM.

[49] Huvert Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2007.

[50] NVIDIA Corp. *CUDA Programming Guide*, 1.1 edition, November 2007.

[51] Travis E. Oliphant. *Guide to NumPy*. Trelgol Publishing, 2008.

[52] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, New York, NY, USA, 2006. ACM.

[53] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, 2003.

[54] Matt Pharr and Randima Fernando, editors. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[55] Mark J. Kilgard Randima Fernando. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Pub Co, 2003.

[56] Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, 2004.

[57] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.

[58] M. Salib. Starkiller: a static type inferencer for python. In *Proceedings of the Europython conference*, 2004.

[59] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide*. Addison-Wesley Proffesional, 5th edition, 2005.

[60] Spirit parser library. http://spirit.sf.net.

[61] Guy Steele. *Common LISP*. Digital Press, 2nd edition, 1984.

[62] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Proffesional, 3 edition, 1997.

[63] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM*, 44(10):95–97, 2001.

[64] David Tarditi, Sidd Purti, and Jose Oglesby. Accelerator: simplified programming of graphics processing units for general-purpose uses via data parallelism. Technical Report MSR-TR-2005-184, Microsoft Research, October 2006.

[65] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby*. Pragmatic Bookshelf, 2nd edition, October 2004.

[66] Mads Torgersen. Querying in C#: how language integrated query (LINQ) works. In *OOPSLA '07*, pages 852–853, New York, NY, USA, 2007. ACM.

[67] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[68] Guido van Rossum. Python tutorial. http://docs.python.org/tut/.

[69] Guido van Rossum. Five-minute multimethods in python. http://www.artima.com/weblogs/viewpost.jsp?thread=101605, May 2005.

[70] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In J.C. Rammings, editor, *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–228. USENIX Association, October 15–17 1997.

[71] Sean Whalen. Audio and the graphics processing unit. www.node99.org/projects/gpuaudio/gpuaudio.pdf, 2005.

Paper I

PyGPU: A high-level language for high-speed image processing

Calle Lejdfors and Lennart Ohlsson
Department of Computer Science
Lund University
Lund, Sweden

{calle.lejdfors|lennart.ohlsson}@cs.lth.se

ABSTRACT

Image processing is an area with many computationally demanding algorithms. When implementing an algorithm the programmer has to make the choice of either using a high-level language, thereby gaining rapid development at the expense of run-time performance. Or, using a lower-level language having higher run-time performance, but also a higher implementation cost. In this paper we present PyGPU, an embedded language that enables image processing algorithms to be written in the high-level, object-oriented language Python. PyGPU functions are compiled to execute on the graphics processing unit (GPU) present on modern graphics cards, a streaming processor capable of speeds more than a magnitude higher than those of current generation CPUs. We demonstrate a number of common image processing algorithms, showing how these can be implemented succinctly and clearly using high-level abstractions, while at the same time achieving high performance.

Published in *Proceedings of IJCSIS '07*, pages 66–81, 2007

1 Introduction

Using a high-level language for writing software comes with many benefits. The code is typically easier to read and understand, making spotting bugs easier. The time spent programming is reduced since the programmer need not worry about low level details such as memory management and data storage formats. In the field of image processing, MATLAB [15] is a popular choice of high-level language. MATLAB is based around an array programming model in which algorithms are expressed on whole images instead of their individual pixels. For example, adding two equal sized images A and B is written simply $A + B$.

The downside of high-level languages is poor performance. Even though the individual operations have efficient implementations, the overall performance is generally not enough for computationally intensive applications such as real-time motion-tracking or high-resolution video post-processing. To overcome this lack of performance it is often necessary to implement the algorithm in a lower-level language, such as C/C++ or FORTRAN, instead. However, this comes at a substantial increase in implementation cost, mainly in terms of programmer effort. Using a third-party image processing library such as Intel's Integrated Performance Primitives (IPP) [9], OpenCV [17], or Mimas [1], that provide optimized versions of standard algorithms, it is possible to reduce this cost somewhat. However, the total implementation cost of using a high-performance, lower-level language is typically much greater than when using a higher-level language.

Recently, there has been increased interest in using the graphics processing unit (GPU) present on modern graphics cards as a computational co-processor. The GPU is a highly specialized processor that provides very good performance. On some problems it is capable of outperforming current-generation CPUs by more than a factor of ten [12]. Programming the GPU is done using specialized languages such as NVIDIA's Cg [14], Microsoft's HLSL [6], or GLSL by the OpenGL ARB [21].

Unfortunately, taking advantage of the performance of the GPU requires expressing an algorithm in terms of graphics primitives such as polygons and textures. Doing this requires intimate knowledge of modern real-time graphics programming. Consequently, implementing image processing algorithms to take advantage of GPU comes at a significant implementation cost, even compared to using lower-level languages.

In this paper we present PyGPU, a language for programming image processing algorithms that run on the GPU. It is implemented as an *embedded language* [8] in the high-level, object-oriented language Python [20]. PyGPU using a point-wise image abstraction that, together with the high-level features of Python, allows image processing algorithms to be expressed at a high level of abstraction. By using the GPU for execution, PyGPU is able to achieve performance in the order of 2–16 GFLOPS without optimizations even on mid-range hardware. This is more than enough to perform real-time edge-detection, for instance, on high-definition video streams.

The rest of this paper is organized as follows: In Section 2 we introduce PyGPU and show a number of example image processing-related algorithms. In Section 3 we discuss performance considerations. Section 4 contains an overview and discussion of PyGPU and how the restrictions and capabilities of the GPU affect how algorithms are implemented. In Section 5 we summarize the contributions made in this paper.

2 PyGPU

PyGPU is a domain-specific language for image processing with a compiler that can generate code which executes on the GPU. It is implemented as an embedded language in Python. An embedded language is constructed by inheriting the functionality and syntax of an existing *host* language. This enables PyGPU to get a lot of high-level language features for free. Python, with its dynamic typing and flexible syntax, allows the embedding to be made very natural manner. Furthermore, using the extensive reflection support of Python, the PyGPU compiler can be implemented very concisely as described in [13].

The fundamental abstraction in PyGPU is its image model. An image is modeled as a function from points on a 2-dimensional discrete grid to some space of colors (RGB, YUV, gray scale, CMYK, etc). As will be shown, this functional model admits expressing image processing algorithms concisely using the high-level language constructs of Python. Also it has the advantage of mapping naturally to the capabilities and restrictions of the GPU.

Below is a small PyGPU function implementing a simple skin detector. It uses the fact that the color of human skin typically lies within a bounded region in the chrominance color plane:

```
@gpu
def isSkin(im=DImage, p=Position):
    y,u,v = toYUV(im(p))
    return inRange(u, uBounds) and \
           inRange(v, vBounds)
```

Looking at the function we see that it has a decorator named `@gpu`. This is a directive to PyGPU's compiler to generate code for the GPU for this function. The default values, `DImage` and `Position`, are type-annotations that are required to compile the function for the GPU.

Apart from these details the function looks like ordinary Python code. The function body shows that to determine if the pixel `p` contains skin we first transform the color value of the pixel `p` in the image `im` to the YUV color space. Then we check if the red and blue chrominance values `u` and `v` both lie within the specified bounds.

Applying the skin detector to an image is done by calling it as an ordinary Python function:

```
skin = isSkin(hand)
```

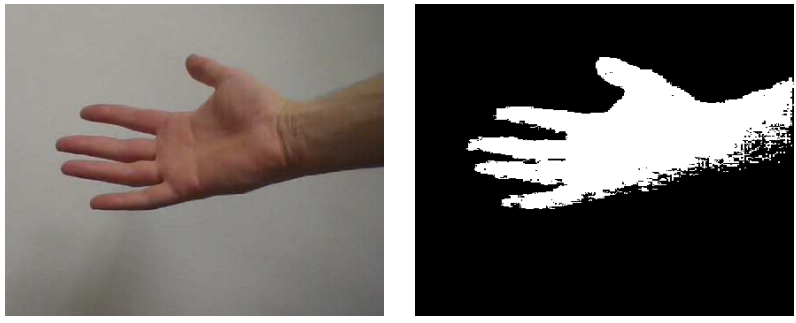


Figure 1: Skin detection

Note that the position argument is omitted, the skin detector is applied to the whole image. The result is shown in Figure 1.

The functions toYUV and inRange are examples of functions from the standard library of PyGPU. This library also provides standard mathematical operations such as basic arithmetic operators, trigonometric functions, and logarithms. These operations work on both scalars and, element-wise, on vectors. PyGPU provides vectors of dimension two, three, or four. Vector operations such as scalar products, and multiplication by scalars are provided through operator overloading, giving an obvious semantics to an expression such as

```
v + a
```

where v is some vector and a either a vector or a scalar.

2.1 Convolutions

The skin detector is an example of the most basic kind of image operations where each pixel in the result image only depends on the pixel at the same position in the sources image(s). Many algorithms, however, require access to multiple source image pixels to compute a single pixel in the result image. Convolution operations, such as differentiations and filters, are typical examples of such algorithms. One example of a convolution is the Sobel edge detector seen below. The edge strength of a pixel is determined as the length of an approximation of the image gradient.

```
@gpu
def sobelEdgeStrength(im=DImage, p=Position):
    Sx = outerproduct([1,2,1], [-1,0,1])
    Sy = transpose(Kx)
    return sqrt(convolve(Sx, im, p)**2 + \
                convolve(Sy, im, p)**2)
```

The gradient is estimated by the convolution of the so called Sobel kernels, one for the horizontal and one for the vertical direction. One can conveniently be expressed as the outer product of two vectors and by symmetry the other is the transpose of the first one.

This example shows a particularly powerful aspect of PyGPU. The functions transpose and outerproduct are not PyGPU functions but come from Numarray, an established high performance Python array programming library implemented in C [7]. And yet these functions can be used in code that is compiled for the GPU. The reason this works is that the compiler uses generative techniques [3] to partially evaluate the code at compilation time [13].

In addition to allowing the use of third-party extension libraries, this generative feature makes it possible to use high-level language constructs such as lists and list comprehensions or built-in standard Python functions even though these features cannot be directly translated to the GPU. For example, the convolve function used above can be succinctly expressed as:

```
def convolve(kernel, im, p):
    return sum([w*im(p+o)
                for w,o in zip(ravel(kernel),
                               offsets(kernel))])
```

The Numarray function ravel is used to compute the column-first linearization of the kernel. Using the built-in Python function zip to combine each kernel element with its corresponding offset (computed by the offsets helper function), the list of weighted image values can be expressed as a list comprehension. The final result is then computed by the standard Python function sum.

2.2 Iterative algorithms

The operations presented thus far have been algorithms where the result is computed in a single pass. Many operation use an iterative strategy where successive applications gradually improve the quality of the result. One example of such an algorithm is anisotropic diffusion filtering [19] that allows efficient removal of noise without simultaneously blurring edges in an image. One step of Perona-Malik anisotropic diffusion can be expressed as

```
@gpu
def pmAniso(edge=DImage, im=DImage, p=Position):
    offsets = [(1,0), (-1,0), (0,1), (0,-1)]
    return im(p) + 0.25*sum([f(edge, im, p+dp, p)
                             for dp in offsets])
def f(edge, im, x, p):
    return g(0.5*(edge(x)+edge(p)))*(im(x)-im(p))

def g(x):
    return e**(-x/(K*K))
```

Figure 2: Perona-Malik anisotropic diffusion.

The function pmAniso is the main function that is compiled for the GPU and the functions f and g are helper functions which are generatively evaluated during the compilation process. The function g controls the conduction coefficients of the diffusion process with K determining the slope. The choice here is one of the functions used in the original paper.

Iteratively applying the diffusion operator to an image can either be done by the standard PyGPU function iterate or by direct loop as shown below:

```
edges = edgeStrength(im)
for i in range(n):
    im = pmAniso(edges, im)
```

This results in successively more smoothed versions of the original image. Figure 2 shows an example image and the result of applying 400 iterations of the anisotropic diffusion operator using $K = 0.25$.

2.3 Reductions

One common pattern in the above examples is that the result of the operation is always another image. In image analysis, however, it is often the case that the result of an operation is instead some overall property of the image, for example the maximum or average image color. These kinds of operations are called *reductions*, operations which reduce the size of an image down to a single value or set of values. For example, a function which computes the pixel-wise sum of an image can be implemented as:

```
def sumIm(im):
    return reduceIm(add, im)
```

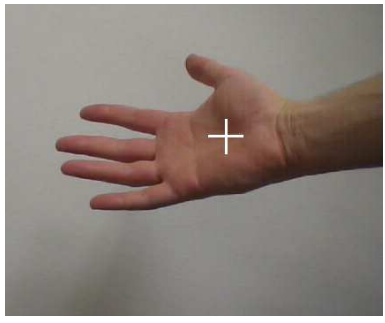


Figure 3: Center of mass

Here, the function add is passed as an argument to a general reduceIm operation. This function is provided by PyGPU and works analogously to Python's built-in reduce but on 2-dimensional images instead of on lists. It is implemented as an iterative algorithm similar to the example in the previous section. Its implementation will be shown in Section 2.6.

A useful example of a reduction is the calculation of the center of mass of a region in a binary image. It can be used, for instance, to approximate the center of a hand or face detected by the skin detector above. The center of mass is the average position of all pixels in the region and can be computed as:

```
def centerofmass(im):
    return sumIm(pos(im))/sumIm(im)

@gpu
def pos(im=DImage, p=Position):
    return p*im(p)
```

The result of applying the center of mass detection algorithm to the result of the skin detector above can be seen in Figure 3.

2.4 Multi-grid operations

One of the advantages of programming in high-level languages is that the abstraction mechanisms available makes it possible to package complex operations as basic building blocks that can be used to construct even more complex operations. As an example we will show the implementation of an operation from the notion of *Poisson editing* introduced by Pérez, Gangnet, and Blake in [18]. The example is called seamless cloning and it is a technique for pasting parts of one image into another in such a way that there is no visible seam between the two images. The idea is to solve the Laplace equation for both images and only replace the differences from these solutions in the pasting operation.

The Laplace equation states that the sum of the second derivates should be equal to zero. In the case of discrete images this is equivalent to saying that a pixel should be equal to the average of its four nearest neighbors. This average is computed by the following PyGPU function.

```
@gpu
def crossAverage(im=DImage, p=Position):
    offsets = [(1,0), (−1,0), (0,1), (0,−1)]
    return sum([im(p+o) for o in offsets])/4
```

Using the standard higher-order PyGPU function masked, that applies a function within a given mask and leave the values outside unchanged, we can express one part of the Laplace equation solver as:

```
x = masked(crossAverage, m)(x)
```

The statement is of the same form as in the anisotropic diffusion example above. It can be used as the basic step in an iterative solver where each iteration yields a successively better solution. The complete implementation of seamless cloning can be expressed succinctly as:

```
def solveLaplace(x, mask):
    return iterate(n, masked(crossAverage, mask), x)

def seamlessCloning(source, target, mask):
    source0 = solveLaplace(source, mask)
    target0 = solveLaplace(target, mask)
    return = (source−source0) + target0
```

An example of seamless cloning can be seen in Figure 4.

The Laplace solver above will eventually reach a solution, but it converges very slowly. For the example in Figure 4 it requires on the order of 10 000 iterations to compute source0 and target0, respectively. A standard technique to improve convergence is to use a *multi-grid* approach where solutions are first found at a lower resolution. This approximate solution is then used as input to solving the problem at the higher resolution level, giving a better initial value for the solution and thereby achieving faster convergence. By changing the definition of solveLaplace to

```
def solveLaplace(x, mask):
    return maskedMultigrid(n, crossAverage, mask, x)
```

The example instead converges in around 200 iterations. The maskedMultiGrid solver is available in the standard library of PyGPU. Its implementation will be shown in Section 2.6.

2.5 Sparse operations

The kind of image operations where the parallelism of the GPU is most efficiently used are *dense* operations, where the computations involve all pixels in the image. All

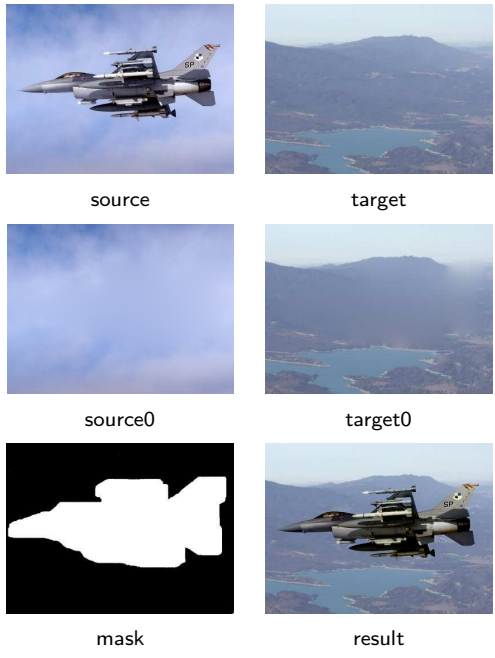


Figure 4: Seamless cloning

operations we have shown so far are all examples of this kind. *Sparse operations* on the other hand operate only on a well chosen subset of points in the images, for example feature points such as detected corners. The irregular access pattern used by sparse methods make them less suitable for implementation on the GPU.

Some kinds of operations use a combination of dense and sparse methods. One class of such operations are active contours or snakes [11] where a polygon is used to define an image area that is interesting in some sense. The contour can automatically search for its area by iteratively moving the polygon until a local minimum is found on a suitably defined *energy function*. This function typically consists of a weighted average of two separate components: the internal energy and the external energy. The external energy is a measure of the image being analyzed, whereas the internal energy is a measure of the shape of the contour itself, for example its smoothness.

The idea is to sample the neighborhood of each vertex of the snake and if any position in this neighborhood gives the vertex a lower energy it is moved to this position. This step is then repeated as many times as needed. A simple implementation of active contours is:

```
def externalEnergy(im, vs, o, v):
```

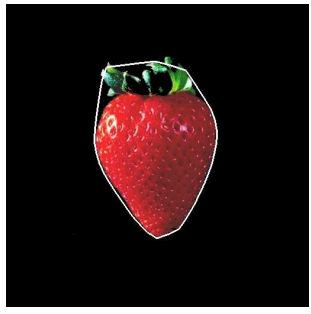


Figure 5: Contour detection using the snake algorithm

```
        return im(vs(v)+o)[0]

    def internalEnergy(vs, o, v):
        p,x,n = [vs((v+i)%nVerts)[0:2]
                 for i in [-1,0,1]]
        x += offset
        m = (p+n)/2
        return norm(x-m)/norm(p-m)

    def totalEnergy(wInt, wExt, im, vs, o, v):
        return wInt*internalEnergy(vs, o, v) + \
               wExt*externalEnergy(im, vs, o, v)

    @gpu
    def energyOptimize(wInt=Float, wExt=Float,
                       im=DImage, vs=DImage, v=Int):
        offsets = array([[0,0],
                         [1,0], [-1,0],
                         [0,1], [0,-1]])
        energies = [totalEnergy(wInt,wExt,im,vs,v,o)
                    for o in offsets]
        return vs(v) + min(zip(energies, offsets))[1]
```

Here, the parameters im and vs contain the image we are optimizing over and the vertices of the polygon, respectively. The weights wExt and wInt contain the relative weights of the external and internal energy. The use of min relies on the fact that comparison between tuples in Python is defined lexicographically. This means that we will find the energy minimum since this is the first member in each tuple. The corresponding offset of that energy minimum is given as the second tuple entry.

The input image used for the external energy is typically not the image being analyzed but rather some preprocessed version, for example a segmented version with edge enhancements. The internal energy shown here is simply a measure of how far a position is from the midpoint of the two neighboring vertices. This choice will give

Figure 6: Block-wise reductions

a "rubber band"-like snake contour where a enclosed region is always convex. Many other variants are possible. The result of applying the snake algorithm is shown in Figure 5.

2.6 Implementation of some generic operations

In the previous sections we have used some generic high-level operations such as reduceIm and maskedMultigrid. Although these are very general and powerful, their implementation in PyGPU is still fairly simple.

The reduction operator is implemented by successively applying the base operation to blocks of the image, resulting in smaller and smaller intermediary results. When the size of the image is 1×1 it will contain the sought quantity as illustrated in Figure 6. For a square image having sides that are a power of two, the operation can be implemented in PyGPU as:

```
block = array([(0,0),(0,1),(1,0),(1,1)]

def reduceIm(f, im):
    @gpu
    def _reduce(im=DImage, p=Position):
        return f([im(2*p+o) for o in block])

    while im.size[0] >= 1 and im.size[1] >= 1:
        im = _reduce(im, _targetSize=im.size/2)

    return im
```

This inner function, which is the one executed on the GPU, successively applies the function f to 2×2 blocks of the image im until it is reduced to a single 1×1 image. The actual reduction in image size is achieved by the parameter _targetSize which is implicitly made available on all PyGPU compiled functions with a default value of the size of the input image.

	# pixel ops.	# texture accesses	Gpixel ops./s	Tex. reads (GB/s)
Convolve (3×3)	27	9	0.56	3.0
Convolve (7×7)	151	49	0.65	3.4
Skin detection	57	1	3.9	1.1
Anisotropic diff.	43	10	0.58	2.1
Laplace solver	18	4	0.60	2.8

Table 1: Performance figures for some of the examples.

A multi-grid solver first finds an iterative solution on a coarse resolution of the image which is then used as the initial value on successively finer resolutions. This masked multi-grid solver in PyGPU can be expressed as:

```
def maskedMultigrid(n, f, mask, x, minSize):
    y = None
    for x, m in reversed(zip(averageR(im, minSize),
                             averageR(mask, minSize))):
        if not y: y = x
        else: y = masked(inflate(y), m)(x)
        y = iterate(n, masked(f, m), y)
        return y
```

The averageR helper function generates a sequence of successively coarser representations of an image down to size minSize. The function inflate does the opposite, *i.e.*, it computes the input to the next higher resolution level.

3 Performance

Although the compiler of PyGPU does not yet implement a number of important optimizations it typically achieve between 0.5 and 4 GPixel operations per second (roughly equal to 2 to 16 GFLOPS) on the examples shown in this paper. This means that a 9-tap convolution filter can be applied to a 500×500 RGBA color image in about 13 ms. The examples were run on a NVIDIA GeForce 6600 graphics card, a lower mid-range card at the time of writing.

Table 1 gives a summary of the performance figures for the most representative examples in this paper. The execution times are essentially proportional to the number of pixels times the number of instructions in the compiled shader program to execute for each pixel. They also include a constant overhead for each pass for setting up the graphics cards, passing parameters to the GPU program, and constructing the result texture. This overhead corresponds roughly to the computation of a couple of thousand pixels, meaning that it is negligible for larger images.

The theoretical peak performance of the NVIDIA 6600 card of our test setup

is approximately 4.8 GPixel operations per second (300 MHz core clock, 8 pixel pipelines using instruction co-issuing) with a peak memory bandwidth of 4 GB/s (500 MHz bus clock, 128 bit bus bandwidth, 64 bits per memory access). As we see from the performance figures, programs that perform more computations relative to the number of texture accesses per pixel perform very well. For example, the skin detection algorithm is able to reach 80% of the computational peak performance.

However, programs that perform many texture accesses per computed pixel quickly become bounded by the available memory bandwidth. This is particularly true for the convolution filters that achieve 75% and 85% bandwidth utilization, but with only 11% and 13% computational efficiency, for the 3×3 and 7×7 case, respectively.

These figures indicate that the key limiting factor in many GPU programs is memory bandwidth. At present, PyGPU is not optimized for minimizing bandwidth consumption. For example, all computations are carried out on 32-bit floating point 4-tuples, which means that both gray scale and binary images are treated as full four channel RGBA images. By using more compact storage formats, as well as reducing the precision to 16-bits where possible, the bandwidth requirements will be reduced and performance increased further. These improvements, as well as trying to locate other bottlenecks in the processing pipeline, are things which will be incorporated in future versions of PyGPU.

4 Discussion

As we have seen the PyGPU language combines high-level programmability with high performance. Being embedded in Python allows functions running on the GPU to be called transparently from Python, greatly facilitating integration of GPU algorithms in larger applications. Furthermore, since PyGPU functions are, at the same time, valid Python functions GPU programs can be tested on the CPU before being run on the GPU. This allows standard debugging and testing tools to be used for GPU programs also, reducing the need for more specialized GPU debugging tools [4].

The performance of the GPU comes from it having a pipelined, highly parallel architecture. This introduces a number of restrictions on what kinds of operations are possible to implement on the GPU. It lacks writable memory. Memory is read only and may only be accessed only in the form of textures containing up to 4-tuples of floating point values. This means that Python features such as lists and objects cannot be used directly on the GPU. But, as we have seen, they may be used to construct programs. For information on how this is achieved, see [13].

Also, GPU programs can only write output to a predetermined image location. This means that GPU algorithms must be, using the terminology of parallel computation, written using a *gather*, rather than *scatter*, approach. This restriction is encoded in PyGPU's image model, where algorithms are expressed in a point-wise manner using only gather operations. This is also the reason why the general reduce operator,

used to do summation for example, is implemented as a iteration over a sequence of progressively smaller images, rather than using a straightforward accumulation loop.

This lack of scatter support sometimes creates difficulties. One such problematic example is computing histograms. This operation is traditionally implemented as a loop over all pixels, having time-complexity linear in the number of pixels. Since the GPU does not support for scattered writes it must instead be implemented as a reduction

```
histogram = reduce(countBins, toBins(im))(0,0)
```

where toBins sorts pixels to their respective bins and countBins count the number of occurrences in each bin. GPUs only support outputting a limited number of values per pixel, currently 16 floating point values. With a larger number of bins than this the algorithm must be run multiple times resulting in a time-complexity on the order of the number of pixels times the number of bins. This illustrates that not all kinds of image processing algorithms are suitable for the GPU.

4.1 Related work

PyGPU was inspired by Pan written by Elliott *et al.* [5], which is an domain-specific language for image synthesis embedded in the function language Haskell [10]. In particular, the functional image model of PyGPU is very similar to that of Pan, but where Pan uses a smooth model, PyGPU focuses on a discrete formulation that allows easier pixel-wise addressing for operations such as convolutions *etc.*

Other domain-specific languages for using the GPU as a computational co-processor have been proposed. For example, BrookGPU by Buck et al. [2] is a compiler for writing numerical GPU algorithms in the Brook streaming language, an extension of ANSI C that incorporates *streams* and *kernels*, representing data and operations on data, respectively. The stream and kernel primitives can be mapped to efficient programs running on the GPU. Also, Sh by McCool *et al.* [16], for instance, uses C++ templates to provide stream processing abstractions similar to those of Brook. These two projects are based on C and C++, respectively. By using Python, PyGPU is able provide higher-level facilities for writing GPU image processing algorithms than currently possible with these approaches.

4.2 Future work

The current syntax of PyGPU requires the programmer to clearly make the distinction between the parts of the code that should execute on the GPU and the parts that should executon the CPU. A nice feature would be to have the compiler be able to do this allocation by itself. Apart from relieving the responsibilities of the programmer, it would also allow the compiler to perform more optimizations, both on for storage requirements and also load-balancing.

Also, in order to translate a Python function to the GPU, PyGPU's compiler must know the types of the function parameters. Currently, this information must be provided by the programmer. An interesting improvement would be to remove this requirement and instead have the compiler automatically infer the necessary type information.

5 Summary

We have presented PyGPU, a language for image processing on the GPU embedded in Python. The functional programming model used by PyGPU allows algorithms to be translated to efficient code running on the GPU, while still retaining the high-level language features allowing them to be implemented concisely and clearly. The performance of PyGPU is good, allowing many algorithms to be run on real-time streaming video sequences without need for special optimization. This enables the implementor to receive rapid feed-back during algorithm development and debugging.

Also, by using language embedding the high-level benefits of Python are transferred onto PyGPU, allowing features such as list comprehensions and higher-order functions to be used in the construction of image processing algorithms. By writing at a higher level of abstraction the code is easier to read and understand. Furthermore, constructing more complex algorithms from simpler building blocks facilitates error detection, making algorithm development and implementation faster and easier.

Bibliography

[1] Bala Amavasai. Mimas toolkit. http://www.shu.ac.uk/mmvl/research/mimas/.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[4] Nathaniel Duca, Krzysztof Niski, Jonathan Bilodeau, Matthew Bolitho, Yuan Chen, and Jonathan Cohen. A relational debugging engine for the graphics pipeline. *ACM Trans. Graph.*, 24(3):453–463, 2005.

[5] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27. Springer-Verlag, 2000.

[6] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[7] Perry Greenfield, Jay Todd Miller, Jin chung Hsu, and Richard L. White. numarray: A new scientific array package for python. PyCon DC 2003, March 2003.

[8] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4):196, 1996.

[9] Intel integrated performance primitives. http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/.

[10] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. ISBN: 0521826144.

[11] Michael Kass, Andrew Witkin, and Demetri Terzopolous. Snakes: Active countour models. *International Journal of Computer Vision*, pages 321–331, 1988.

[12] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[13] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded GPU language by combining translation and generation. *Proceedings of SAC'06*, 2006.

[14] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[15] Matlab. http://www.mathworks.com/.

[16] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[17] Open source compter vision library. http://www.intel.com/technology/computing/opencv/.

[18] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, 2003.

[19] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.

[20] The Python language. http://www.python.org/.

[21] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.

Paper II

Implementing an embedded GPU language by combining translation and generation

Calle Lejdfors and Lennart Ohlsson
Department of Computer Science
Lund University
Lund, Sweden

{calle.lejdfors|lennart.ohlsson}@cs.lth.se

ABSTRACT

Dynamic languages typically allow programs to be written at a very high level of abstraction. But their dynamic nature makes it very hard to compile such languages, meaning that a price has to be paid in terms of performance. However under certain restricted conditions compilation is possible. In this paper we describe how a domain specific language for image processing in Python can be compiled for execution on high speed graphics processing units. Previous work on similar problems have used either translative or generative compilation methods, each of which has its limitations. We propose a strategy which combine these two methods thereby achieving the benefits of both.

Published in *Proceedings of SAC '06*, pages 1610–1614, 2006.

1 Introduction

In this paper we introduce PyGPU, a domain-specific language for writing image processing algorithms embedded in the interpreted, object-oriented, dynamically typed language Python. The PyGPU language consists of a number of classes that allow image processing algorithms to expressed clearly and succinctly. These classes use overloading to provide operations such as multiplying a color by a scalar, and accessing an image, with intuitive semantics. For instance, a function for multiplying every pixel of an image by a scalar can be implemented as:

```
def scalarMul(c=Float, im=Image, p=Position):
    return c*im(p)
```

Furthermore, this function can be compiled to native code executing at very high speeds on the graphics processing unit (GPU). However, as will be described below, the GPU is a very restricted platform, and in order to compile a function type-annotations, as in the above example, are required. The types used are exactly the above classes which here serve the alternate purpose of encoding the restrictions and capabilities of the GPU.

The outline of the rest of this paper is as follows. In Section 2 we introduce the graphics processing unit (GPU) as well as a more extensive example of using PyGPU. In Section 3 we present the implementation of PyGPUs compiler and in Section 4 we finish up with a discussion.

2 GPUs

Most computers come equipped with a powerful 3D graphics card capable of transforming, shading, and rasterizing polygons at speeds in excess of those provided by the CPU alone. These cards are also equipped with a programmable graphics processing unit (GPU) that enable parts of the polygon rasterization process to be programmatically redefined allowing, for instance, many image processing algorithms to be implemented. And, since floating-point performance of the GPU is typically an order of magnitude higher than that of a corresponding CPU [9] it is a very attractive target platform.

The speed advantage of GPUs comes from their highly specific nature; they employ a number of very long pipelines executing in parallel and in order to efficiently use this parallelism the computational model of the GPU is very restricted. There is no dynamic memory allocation. Memory is read-only and may only be accessed in the form of textures containing 4-dimensional floating-point values. Furthermore, flow control structures such as branches and subroutines are not guaranteed to exist even on very modern cards.

Figure 1: Sobel edge detected lena

2.1 Existing GPU languages

There are a number of specialized language available for programming the GPU. The first generations of GPUs provided limited forms of programmability trough assembler-like languages specific to each vendor and graphics API. With the increased power and maturity of GPUs a number of higher level languages were introduced: Cg by NVIDIA [11], HLSL by Microsoft [5], and GLSL by the OpenGL ARB [8]. These languages are all syntactic variants of C with some added constructs and data-types suitable for expressing GPU programs.

Other projects aimed at providing embedded languages for programming the GPU are Vertigo [3] and Sh [12]. Vertigo uses Haskell [7] to program the vertex shader functionality of GPUs. Sh is embedded in C++ and uses a generative model for constructing GPU programs at run-time from specification written in C++. Sh also supports, through the use of C++ templates, combining GPU program fragments into new GPU programs [13].

2.2 An image processing example

We will now provide an extended example of PyGPU by implementing an edge detection algorithm. We will construct a general edge detector which will then be used to implementing the well known Sobel edge detector.

Edge detection in general

Edge detection is the process whereby sharp gradients in image intensity are identified. In general, this can be implemented as the application of convolution kernels estimating the image intensity gradient in the x and y-directions, respectively. Consider the following function definition:

```
def edgeDetect(kernel, im=Image, p=Position):
```

```
Gx = convolve(kernel, im, p)
Gy = convolve(transpose(kernel), im, p)
return sqrt(Gx**2 + Gy**2)
```

This function applies an arbitrary kernel in the x and y-directions (by symmetry the vertical gradient approximation kernel is the transpose of horizontal approximation kernel) and then computes the magnitude of the image gradient.

Gradient approximation kernels

There are many examples of gradient approximation kernels. One common choice is the Sobel operator which can be represented by the matrices

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

for the x and y-directions, respectively.

Complete edge detector

Using the general edge detection function and the kernel from the previous section we can now create a Sobel edge detector by partially specializing the general edge detection function. To do this we call the PyGPU compiler passing the kernel as a compile-time parameter:

```
sobelEdgeDetGPU = pygpu.compile(edgeDetect, kernel=sobelKernel)
```

The function returned by the compiler runs entirely on the GPU and can be applied to images just as a normal function. However, the position parameter need not be specified, the returned function operates on whole images in parallel:

```
edgesLena = sobelEdgeDetGPU(lena)
```

The result of applying the Sobel edge detector to the standard Lena example image can be seen in Figure 1.

Example discussion

Interestingly, the general edge detection function presented above can not be translated to native code on the GPU and the reason lies in the kernel argument. Because the GPU lacks support for dynamic memory allocation translating the kernel argument to the GPU is impossible. PyGPU expresses this by the fact that the kernel argument cannot be given a type in PyGPUs type system. And, since these types encode the capabilities of the GPU, an argument which cannot be typed must be supplied as a value at compile-time.

As a consequence we are allowed to use external libraries even when these libraries cannot be translated to the GPU. For instance, the transpose function is taken directly from Python Numeric, an array programming library implemented in C and running on the CPU[14]. Clearly, this function cannot be directly translated but, since the value of kernel must be supplied at compile-time, it may still be used to construct PyGPU functions.

3 Compiler implementation

The PyGPU compiler is implemented in Python and it is responsible for two major tasks: compiling PyGPU functions to programs running on the GPU, and providing the necessary glue-code allowing these programs to be called as ordinary Python functions. The implementation of the latter is straightforward and will not be covered. The implementation of the translation from Python functions to GPU programs is the focus of this section.

3.1 Related work

PyGPU lies at the intersection of two problem areas related to compilation: dynamic languages and embedded languages. It shares a number of problems from both areas all of which must be overcome to allow effective compilation. Furthermore, the restrictions of the target platform greatly affects implementation choices.

Compiling dynamic languages

Compiling dynamic languages is, in general, a very difficult problem. Most of what we know from static languages cease being true: function implementations can be changed at run-time, arbitrary code can be executed via eval, and classes can be dynamically constructed or changed. One approach is to restrict the dynamism of the language. This is used in PyPy [15] and Starkiller [17], two projects targeted at compiling Python. Both projects perform static analysis such as type inferencing to translate general Python code into lower-level compilable code.

Alternatively, the dynamism can be kept by performing *run-time specialization* to compile functions at call-time. This is the approach taken by Psyco [16], a just-in-time compiler for Python.

Compiling embedded languages

By construction, embedded languages can typically be compiled by the host language compiler. The problem with compiling embedded language however is that they typically target a *different* platform than that supported by the host language. Some

examples of such platforms are co-processors [12], VHDL designs [1], and midi sequencers [6].

The most direct approach for implementing an embedded language compiler is to view the host language merely as syntax for the embedded language. A traditional compiler can then be implemented by reusing the front-end for the host language and implementing a new back end. Such *translative* methods work well when the features of the embedded language closely match the capabilities of the target platform. In such cases translative methods can be implemented fairly directly.

An alternate approach is to use the overloading capabilities of the host language. By implementing a suitable set of abstractions it is possible to *execute* a program in the embedded language in such a way that it generates a program on the target platform. These types of *generative* methods are typically straightforward to implement since much of the existing compiler infrastructure is reused. They are however restricted to translating only those features of the host language that can be overloaded. Conditionals, loops, and function calls, for instance, cannot be overloaded in most languages and consequently cannot translated using this approach. Examples of projects using a generative approach are Pan [4], Vertigo [3], and Sh [12]. Pan and Vertigo are Haskell domain-specific embedded languages for writing Photoshop plugins and vertex shaders, respectively. Both use a tree-representation constructed at run-time to generate code for their respective platforms. Sh is a GPU programming language embedded in C++ that uses overloading to record the operations performed by a Sh program. This "retained" operation sequence is then analyzed and compiled to native GPU code. We will use a combined approach giving the benefits of both these methods.

3.2 Combining translation and generation

Given that we use Python as host language for PyGPU we are faced with a difficult decision. The restrictions of the GPU makes direct translation of features such as lists and generators impossible, requiring either restricting the languages or implementing advanced compiler transformations. Using a generative method we are required to supply our own conditionals and loop-construct thereby sacrificing the syntactic brevity of our host language. Ideally one would like to use a translative approach for those features that admit direct translation and a generative approach for those that do not.

We propose that this can be achieved by combining two features commonly found in dynamic high-level languages: *introspection* and *dynamic code execution*. Introspection is the ability of a program to access and, in some cases, modify its own structure at run-time. Dynamic code executing allows a running program to invoke arbitrary code at run-time. For instance, we can use the introspective ability of Python to access the bytecode of a function, where elements such as loops and conditionals are directly

represented. This allows using a translative approach where possible. Using dynamic code execution we can reuse large parts of the standard Python interpreter to thereby giving the benefits of translative methods.

3.3 The compilation process

As explained above (see Section 2.2) PyGPU requires that types of all free variables are known at compile-time. Parameter which cannot be given a type must be supplied by value. Hence, for every parameter of a function we know either its type or its value. The compilation strategy thus becomes: if the value is known we evaluate generatively, if only the type is known we perform translation.

The compiler is implemented in the usual three stages: front end, intermediate code generation, and back end. The intermediate code generation and back end stages are implemented using well-known compiler techniques. We use static single-assignment (SSA) [2] for representing the intermediate code. This enables many standard compiler optimization, such as dead-code elimination and copy propagation, to be implemented effectively. The optimized SSA code is then passed to a back end native code generator. At the moment we use Cg [11] as a primary code generation target allowing optimizations of that compiler to be reused.

The front end however, differs from the standard method of implementing a compiler. Instead of using text source code it operates directly on a bytecode representation and it is the front end that implements the above compilation strategy. How this is implemented using the dynamic code execution features of Python will now be described in detail.

Bytecode translation

The front end parses the stack-based bytecode of Python and translates it to a *flow-graph* which is passed to the intermediate code generator. Throughout this process the types of all variables are tracked allowing the compiler to check for illegal uses as well as performing dispatch of overloaded operations.

Simple opcodes, such as binary operations, are translated directly. More complicated examples such as function calls, that would not be translatable using a generative approach, are handled using the above strategy:

```
elif opcode == CALL_FUNCTION:
    args = stack.popN(oparg)
    func = stack.pop()
    if isValue(args):
        stack.push(func(*args))
    else:
        compiledF = compileFunc(func, args)
        result = currentBlock.CALL(compiledF, args)
        stack.push(result)
```

That is, if all the arguments are values then the function is evaluated directly in the standard interpreter. This is done by using the dynamic code execution abilities of the standard interpreter to call the function via func(*args). This allows the PyGPU compiler to reuse functionality present in external libraries (even compiled ones) generatively. Note that, in general this kind of constant-folding of function calls is not permitted. The function being called may depend on global values whose value may change between invocations. But, since the GPU lacks globals variables PyGPU does not allow global values to be changed after a function has been compiled and consequently this transformation is valid.

If the value of at least one argument is not known then the callee is compiled and a corresponding CALL-opcode is added to the current block of the flow-graph.

This strategy is not restricted to the case of function calls, it can be used to handle loops as well. Consider the fragment

```
for i in range(n):
    acc += g(i)
```

If n is known at compile-time then we may evaluate range(n). Consequently the sequence being iterated over is known and the loop can be trivially unrolled. If n is not known the fragment is translated to an equivalent loop in the GPU. The code for handling loops is similar to that of handling function calls albeit slightly more complicated.

3.4 An illustrative example

The compilation strategy presented above is very straightforward and it is not obvious how this strategy enables us to translate more complicated examples. Consider the implementation of the convolve function used in Section 2.2:

```
def convolve(kernel, im=Image, p=Position):
    return sum([w*im(p+d)
                for w,d in zip(ravel(kernel),
                               offsets(kernel))])
```

The implementation reads: to compute the convolution we first compute the column-first linearization of the kernel using the function ravel. The offset to each kernel element is computed and each offset is associated with its corresponding kernel element. The image is accessed at the corresponding locations and the intensities are weighted by the kernel element. Finally the resulting list of intensities is summed and the result returned.

Note that here we use a number of features which cannot be directly translated to the GPU: the compiled Numeric [14] function ravel, list-comprehensions, and the built-in Python functions zip and sum both which operates on lists. However, using the above strategy compilation proceeds as follows: The value of kernel must be known at compile-time and, consequently, the values of ravel(kernel) and offsets(kernel)

can be computed. Hence the arguments to zip are known which implies that it may, in turn, be evaluated at compile-time. The resulting list is used to unroll the list-comprehension resulting in a known number of image accesses which can be directly translated to the GPU. The code for summing these accesses and returning is generated similarly thereby concluding the translation of the above function.

4 Discussion

We have shown how a compiler for an embedded language can be implemented to combine the advantages of previous methods. By taking advantage of introspection and dynamic code execution features of the host language Python we could implement this compiler very compactly. As an example, The compiler and run-time consists of around 1500 lines of code with the bytecode to flow-graph translator occupying 400 of those lines. By compiling for the GPU, performance in excess to that of optimized CPU code can be obtained. Furthermore, the relative increase in speed for new GPU generations is greater than the corresponding increase for CPUs making the GPU a very attractive platform.

A recent area of research is using the GPU for general purpose computations. Examples of algorithms which have been implemented on the GPU are fluid simulations [10], linear algebra [9], and signal processing [18]. Future work includes extending the PyGPU compiler to allow programming all aspects of the GPU including general purpose numerical algorithms. Also, the presented method ought to be suitable for compiling embedded languages to other platforms such as ordinary CPUs.

Another interesting area for future research is studying how the approach used here integrates with that of PyPy [15]. Of particular interest is reusing parts of the PyPy framework to be able to handle more general examples, including the above general purpose uses of the GPU as well as targeting other platforms.

Bibliography

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, 1998.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, October 1991.

[3] Conal Elliott. Programming graphics processors functionally. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 45–56, 2004.

[4] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27. Springer-Verlag, 2000.

[5] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[6] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[7] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. ISBN: 0521826144.

[8] John Kessenich, David Baldwin, and Randi Rost. The OpenGL shading language. http://developer.3dlabs.com/documents/index.htm, 2003. 3DLabs, Inc Ltd.

[9] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[10] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3d fluid simulation on GPU with complex obstacles. In *Proceedings of Pacific Graphics 2004*, pages 247–256, October 2004.

[11] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[12] Michael McCool, Zheng Qin, and Tiberiu Popa. Shader metaprogramming. In Thomas Ertl, Wolfgang Heidrich, and Michael Doggett, editors, *Graphics Hardware*, pages 1–12, 2002.

[13] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[14] Numerical python. http://numpy.org.

[15] Pypy - an implementation of python in python. http://codespeak.net/pypy/.

[16] Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, 2004.

[17] M. Salib. Starkiller: a static type inferencer for python. In *Proceedings of the Europython conference*, 2004.

[18] Sean Whalen. Audio and the graphics processing unit. www.node99.org/projects/gpuaudio/gpuaudio.pdf, 2005.

Paper III

Unified GPU programming

Calle Lejdfors and Lennart Ohlsson
Department of Computer Science
Lund University
Lund, Sweden

{calle.lejdfors|lennart.ohlsson}@cs.lth.se

ABSTRACT

Programmable shading has become an essential part of interactive three-dimensional graphics and today, graphics cards provide powerful graphics processing units (GPUs) capable of executing complex shading algorithms in real time. Programming these GPUs requires, in addition to CPU support code, writing two programs, one to be executed for every mesh vertex and one to be executed for every projected on-screen pixel. This separation gives the programmer great control of the rendering pipeline. However, it also makes GPU programs difficult to manipulate or reuse because the vertex/pixel processing unit allocation must be declared explicitly.

In this paper we show that GPUs can be programmed in a unified manner, without explicitly factoring the program into vertex and pixel parts, and without sacrificing control or performance. The split between vertex and pixel computations is automatically inferred by the compiler without requiring any source code annotations. This process can be controlled, making it possible to directly manipulate the performance and visual quality characteristics of the program. Splitting control can be integrated in an interactive artist's tool or in an off-line analysis algorithm, for instance. Previous approaches to unified programming were restricted to Renderman-like surface and light shaders, thereby excluding a large part of the contemporary uses of the GPU. We extend these methods to handle arbitrary GPU programs, including branches and conditionals.

Technical report, Lund University, LU-CS-TR:2008-244
Extended version of a paper accepted for publication in *Proceedings of CGV '08*.

1 Introduction

Programmable shading — the process of computing the on-screen appearance of a graphical model — has become an essential part of three-dimensional graphics and visualization. It is used to achieve a wide range of visual effects, ranging from lighting models to hardware-assisted particle systems to procedural geometry. Programmable shading was introduced in off-line systems [4, 25] where it was popularized by the Renderman shading language [10].

The first real-time shading systems were custom-built architectures such as the Pixelflow system [20]. Today, graphics cards provide high performance, programmable *graphics processing units* (GPUs) which allow the transformation, tessellation, and pixel processing stages of the rendering pipeline to be controlled. This enables GPU programs, commonly referred to as *shaders*, to combine geometric effects, such as mesh deformations and animation, with advanced lighting, texturing, and colorization computations. Since their introduction in 2001 [17], programmable GPUs have been used to implement a wide range of graphical applications: lighting models [1, 4], procedural materials [29], geometric detail approximation techniques [13], etc. Also, GPUs have been successfully deployed in non-shading applications such as skinning, particle systems, procedural geometry (e.g., water, grass), and image post-processing. All these techniques are today considered an essential part of GPU programming.

When writing GPU programs, the programmer must typically specify at least three different programs: one running on the host CPU for interfacing with the application, and the vertex and pixel shaders, for doing calculations on the GPU. The vertex shader executes early in the pipeline and is responsible for computing geometric data such as position, normals, and texture coordinates. The pixel shader then uses the interpolated geometric data to compute the final pixel color. This separation stems from the structure of the graphics pipeline and it gives the programmer great control over both visual quality and performance. However, it also makes GPU programming needlessly complex since the programmer must focus not only on implementing the algorithm correctly, but also distributing the computations across the vertex and pixel processing units. Furthermore, explicitly splitting what is essentially a single operation (that of transforming and shading a mesh) makes modification, e.g. by changing from per-vertex normals to normal maps, more difficult.

In this paper, we show that GPUs can be programmed in a unified manner, allowing both CPU, vertex and pixel computations to be specified in a single program. Using this unified approach, GPU programs to be implemented more directly, resulting in programs that are easier to maintain and manipulate. In our approach we stay close to the capabilities of existing hardware in the sense that we require the same degree of control over performance and visual quality as in current GPU languages. In our approach, we are thereby able to extend previous attempts at unified shading [26] to handle general GPU programs including:

- arbitrary vertex computations such as geometry deformations, tangent frame computations, and shadow volume extrusion,
- branches and conditionals in GPU programs,
- vertex texture fetches,
- multiple outputs at different computational frequencies,
- fine-grained type system necessary to support correct interpolation of unit vectors,
- external control of the how the unified program is distributed across the vertex and pixel processing stages of the GPU,

By supporting general, arbitrary GPU programs we enable a large number of complex effects to be implemented, such as, sub-surface scattering [3], model skinning [14], procedural materials [8], ray-marching algorithms [16], particle systems [15], or LOD-branching [28].

Furthermore, being able to externally control how a program is distributed across the GPU enables fine-grained control of the performance characteristics of that program. More importantly, this control is achieved without requiring any source code modifications. It makes it possible to, for example, shift computations to the vertex unit which can result in performance gains. We present a dependency analysis algorithm capable of automatically splitting unified GPU programs across the CPU and GPU. This algorithm identifies operations which can be computed by either the vertex or pixel processing units. By default such operations are performed per-pixel but this can be controlled via the external interface. We demonstrate how this type of control can be integrated with the artist's tool chain or in an offline analysis tool.

The rest of the paper is organized as follows: in Section 2 we give an overview of related work. Section 3 introduce unified GPU programming in more detail. Section 4 present the splitting algorithms and discusses GPU operation scheduling. Section 5 discusses integration of the external control mechanism and finally, Section 6 summarizes the contributions of this paper and discusses future work.

2 Related work

The first GPU generations were programmed using assembler languages but today, specialized high-level GPU languages, such as Cg [18], HLSL [9], or the OpenGL Shading language [27], are typically used. These languages provide a C-like syntax with constructs and data-types tailored specifically for the GPU. Also, there exist some embedded languages for GPU programming, most notably Sh [19] and Vertigo [6].

A number of works have focused on compiling Renderman, or Renderman-like, surface shaders for real-time graphics hardware. Olano and Lastra [22] compile shaders written in a Renderman-like language to the Pixelflow platform [20]. Peercy et al. [23] compile surface shaders into a series of OpenGL rendering passes, which are combined using framebuffer blending. In the Stanford real-time shading language (RTSL) project [26] the problem of compiling Renderman-like surface shaders to modern programmable graphics hardware was addressed. They combined data-dependence analysis with optional user-annotations to split shaders into parts executing at different *computational frequencies* (constant, model, vertex, and pixel) where the vertex and pixel frequencies map to the vertex and pixel processing units of the GPU.

Olano et al. [21] introduced a technique for reducing and simplifying memory accesses in GPU programs, a key performance-limiting factor in real-time shader programming. The approach used is to pre-compute parts of a shader program, using textures to cache the results. This inspired Pellacini [24] to create a method for the automatic simplification of arbitrary shader program computations. He present a technique based on transforming an abstract syntax representation of the shader to reduce the amount of computations necessary. The resulting reduced shaders are then analyzed using a Monte Carlo sampling scheme, to evaluate the visual error of a reduction. The approach of Pellacini is widely applicable to per-pixel shading, including off-line rendering systems, but does not address splitting computations across vertex and pixel shading units.

Another shader error estimation scheme was proposed by Heidrich et al. [11] who employed affine arithmetic to guarantee error bounds when sampling procedural shaders. This enables a compiler to do optimizations based on error bounds analysis, resulting in faster shading computations without affecting visual quality. This approach is, however, better suited for off-line rendering where the rendering time typically out-weighs the time spent analyzing and specializing shaders.

3 Unified GPU programming

When implementing an algorithm in a GPU program the programmer must typically write three distinct programs. One that executes per-vertex to compute vertex clip-space positions and other geometric information that is interpolated across the primitive. One program executing per-pixel, using the interpolated output of the vertex program to compute the final pixel color. Finally, a CPU program is required to do parameter passing, data loading, user interaction, etc. This division gives the programmer great flexibility, and allows algorithms to be implemented efficiently by using a combination of CPU pre-computations and GPU code. It also enables algorithms to be tailored to meet specific demands for performance and visual quality. For example, a shading algorithm such as Blinn-Phong can be computed entirely per-

pixel, giving high visual quality, or per-vertex, typically giving higher performance at the expense of visual quality. Another variant is a to compute the diffuse term per-vertex and the specular term per-pixel. This approach gives better performance without sacrificing visual quality by computing slowly changing lighting components at a lower computational frequency.

Today, these types of trade-offs are expressed explicitly in the source code by forcing the programmer to write separate programs for each programmable step in the pipeline. This implies that a given GPU program is expressed as a combination of intimate knowledge of the target hardware and the application-specific performance and visual quality demands. So depending on whether a particular algorithm is to be applied on a finely or coarsely tesselated mesh, it can be implemented differently. On a finely tesselated mesh one can use more vertex computations since the relative error when interpolating across the primitives will be small. When used on coarser meshes a greater degree of pixel computations are required to avoid shading artifacts. Also, since such considerations are represented explicitly in the code, adapting or reusing such a shader becomes a needlessly complex and time-consuming task.

We argue that information regarding how a program should be distributed across the processing stages of GPU should not be explicit in the program itself. Instead it should be supplied via an external mechanism, enabling this distribution to be controlled without rewriting the original program source code. In this paper, we show how a GPU program can be automatically distributed across the vertex and pixel processing units while retaining numerical precision. Operations which can be computed in either the vertex or pixel processing stage are initially assigned to the pixel stage, to achieve maximal numerical precision, but can be moved to the vertex unit via an external interface. This reduces the number of explicit assumptions expressed in the program source code, making it easier to reuse that program in another application or on another hardware platform. Also, by allowing external modification, GPU programs can be quickly adapted to a particular usage scenario (such as modifying a lighting algorithm to use vertex instead of pixel computations to gain performance at the expense of visual quality). This type of control can be integrated in an artist's tool, for example, and reduces the burden on the programmer, leaving him or her free to focus on producing high-quality algorithms instead of tweaking low-level details.

3.1 A unified shader language

In a unified GPU program, or a *unified shader*[1], shading computations from different computational frequencies are written as a single program. A compiler analyzes this program and automatically factorizes it across the CPU and vertex/pixel processing units of the GPU. In this paper we use a prototype GLSL-like language implemented

[1] Not to be confused with a *unified shading core* which is a hardware unit that can act both as a vertex and as a pixel shading unit.

as an embedded language [12] in Python. This approach enables rapid experimenting with language features without the need for implementing a whole compiler suite (parser, front-end, back-end, etc.) from scratch. The method presented here is not limited to this case, however. The algorithms described below operate at the level of data-dependencies and code blocks, and is stable across different language front-ends.

As in GLSL, GPU programs declare their input parameters together with where these parameters appear; attribute for vertex-data such as position and normals, and uniform for model-data such as textures and transformation matrices. Also, the full range of GLSL built-in parameters are provided: gl_ModelViewProjectionMatrix, gl_Vertex, gl_Normal, and so on (see Rost [27]). The shader defines a function main that computes the color and the final clip-space position which are written to the dedicated variables gl_Position and gl_FragColor, respectively. For an example of a unified shader, see Listing 1 (page 65). This shader implements animated water using the approach of Finch [7], by combining Gerstner waves, and bump-mapped normal perturbations.

4 GPU program factorization

Given a unified GPU program we must split it across the vertex and pixel processing units of the GPU. This problem was first addressed in a restricted form in the Real-Time Shading Language (RTSL) [26]. That approach was to infer the computational frequency of each operation using data-dependency analysis, where frequency information is propagated forward in the data-flow graph. The inferred frequency is then used to assign the operation to either the CPU, or the vertex and pixel processing stages of the GPU.

In RTSL, the lowest frequency at which it is possible to perform an operation is used. So, if the possible frequencies are (ordered from lowest to highest) *constant*, *model*, *vertex*, and *pixel*, then the frequency of an operation f with arguments $x_1, x_2, \ldots x_n$ is given by the formula

$$F\big(f(x_1, x_2, \ldots, x_n)\big) = \max_{i=1,\ldots,n} F(x_i)$$

This implies that in RTSL the specular term in Blinn-Phong, $dot(H, N)$, for example, is computed per-vertex if H and N are supplied per-vertex. Had N been computed per-pixel, the scalar-product would be computed per-pixel also.

The forward propagation approach works well when restricted to the branch-free light and surface shaders supported by RTSL. However, when generalizing to arbitrary GPU programs with conditionals the algorithm will fail since it does not properly take into account the relative frequencies of the conditional expression and the operations of conditional block (see Section 4.1 below). Furthermore, it is not conservative and will sometimes yield numerically incorrect results. Consider the specular term in the

Blinn-Phong shading equation:

$$specular \leftarrow dot(H, N).$$

Computing this term per-vertex or per-pixel will give very different results. The reason for this is that the scalar product does not vary linearly across the primitive. RTSL solves this by forcing the programmer to explicitly state that the above operation should be computed per-pixel in this case.

We solve both these problems by using a bi-directional frequency-inference algorithm instead. By analyzing the later uses of *specular* we can infer whether it should be computed per-vertex or per-pixel. By using such bi-directional propagation we find that the possible *frequency range FR* of an operation f with arguments $x_1, \ldots, x_n$ and which is used by (i.e. has parents in the data-dependence graph) $p_1, \ldots, p_m$ is given by

$$FR(f) = \left[\max_i \{ \min FR(x_i) \}, \min_j \{ \max FR(p_j) \} \right]$$

This implies that operations whose values will ultimately be used at the vertex frequency (such as vertex deformations, tangent-space computations) must be computed at or below the vertex frequency. Operations which are used to compute pixel data are computed at or below the pixel frequency. Using this formula we can quickly infer the possible frequency range of each operation in the shader.

Our implementation work on a intermediate static single assignment (SSA) representation [5], and if we disregard loops and conditionals for a moment, it proceeds as follows: start at the input nodes (the attributes, uniform, and texture parameters of the shader program) and propagate the minimum of frequency range forward in the dependency graph. This step is similar to that of RTSL. Then, start at the output nodes (pixel color and vertex position) and propagate the maximum frequency range backwards in the dependency graph. This will correctly determine the frequency range of each operation in the program. However, this naive approach will fail if the program contains more than one basic block i.e. if it contains loops or conditionals.

4.1 Handling loops and conditionals

If the program contains a conditional, for example, the above algorithm will not be conservative, meaning that the conditional block might not evaluated at the same frequency as the conditional expression. Consider using alpha-masking to render parts of a diffuse object (typically used to render a chain-link fence or similar structure):

```
alpha = tex2D(maskTex, uv).a
if alpha > threshold:
    gl_FragColor = dot(N,L)*diffuseColor
else:
    discard()
```

Here the naive algorithm may find that the right hand side in the gl_FragColor assignment can be computed either per-vertex or per-pixel. However, the conditional expression must be evaluated per-pixel since it involves a texture look-up which by default are only allowed per-pixel. Hence, the entire conditional must be evaluated per-pixel in this case. Similarly, if the conditional expression may be evaluated either per-vertex or per-pixel, and the conditional block must be evaluated per-pixel, then the entire conditional must be evaluated per-pixel also.

Our algorithm handles this case by propagating the frequency range of the conditional expression e into the conditional block and vice versa. The resulting frequency range of the entire conditional c is given by

$$FR(c) = FR(e) \cap \left(\bigcap_i FR(op_i) \right),$$

where op_i are the operations of the conditional block. Similarly, for loops the frequency range of the loop iteration variable is propagated into the loop block, and the frequency range of the entire loop block is propagated to the loop iteration variable. This implies that a conditional or loop must be scheduled as a unit whose frequency range depends both on the frequency ranges of the conditional expression or loop iteration, and the operations making out the conditional or loop block.

4.2 Operation scheduling

Once the frequency range of all operations is known the operations are either precomputed (for operations with constant frequency) or assigned to either the CPU or GPU. Model frequency operations are assigned to the CPU, and vertex and pixel frequency operations to the vertex and pixel processing units on the GPU. The operation scheduling algorithm is conservative and will, by default, only assign a computation to a frequency which does not introduce numerical errors. So if an operation can be performed at either the vertex or pixel frequency, it is assigned the vertex processing unit only if doing so will not introduce numerical errors. Otherwise it is assigned to the pixel processing unit. This approach correctly handles the above example of the specular term.

Some shaders can be uniquely factored across the processing units of the GPU. For all such shader programs, the unified shader can be directly translated to efficient, native GPU code. In such cases the unified shader approach lets programmers write shaders without explicitly splitting them, allowing simpler implementation and debugging. However, most algorithms can, wholly or in part, be computed either per-pixel or per-vertex. This is indicated by it containing operations that have a frequency range of $[vertex, pixel]$. We call such operations *approximation sites* since they represent computations which can be performed per-pixel, or approximated by computing them per-vertex and interpolating linearly.

Typically, a shader have one or more approximation sites, depending on its type and complexity. For example, a relatively simple shader such as Blinn-Phong can contain as many as 11 approximation sites. Deducing which approximation sites we can compute per-vertex will result in a very large search space (at worst 11^2 different combinations) if we use a naive selection method, whether it is based on user-input or more computationally oriented methods such as Monte Carlo sampling. Thus, to effectively test for suitable approximations of a shader we must use pruning. To do this we use a collection of rules which will conservatively reduce the search space. These rules match the strategies used by shader programmers when writing shaders by hand, hence showing that it is possible to automate and formalize much of this type of domain-specific optimization knowledge.

Linear operations Operation which depends linearly on their arguments are excluded from the set of possible approximation sites. Linear operations are invariant under linear interpolation so if we, for example, have the operation $y \leftarrow 2 \times x$ where x is computed per-vertex. Then, we may compute y per-vertex as well since, under linear interpolation by the operator I we have

$$I(y) = I(2 \times x) = 2 \times I(x)$$

Consequently, an operation which depend linearly will always be computed at the same frequency as its arguments and, hence, it is excluded from the set of approximation sites. Note however, that an operation such as

$$z \leftarrow dot(x, y)$$

may or may not be linear depending on the frequency of x and y. If x and y are computed per-vertex, then z is not linear since

$$I(z) = I(dot(x, y)) \neq dot(I(x), I(y)).$$

Hence, such an operation should be assigned to the pixel-frequency by default. Also, it should not be pruned from the set of approximation sites. However, if x is computed per-vertex and y per-model, then we have $I(z) = I(dot(x, y)) = dot(I(x), I(y)) = dot(I(x), y)$ since y does not vary across primitives. Consequently, in this case z can be computed per-vertex without introducing numerical errors.

Normalization operations Unit length vectors that are computed per-vertex and used per-pixel also require special attention. To correctly interpolate unit vectors across a primitive, two methods must be used. For unit vectors such as normals, tangents, bi-normals, etc., the vector is first computed and normalized per-vertex. Then, after interpolation, the result is normalized again to account for the fact that linear interpolation does not preserve lengths. However, direction vectors such as light and

viewer directions in Blinn-Phong should not be normalized per-vertex! Normalization should only be done after interpolation, in the pixel processor. This is necessary, for example, to ensure that the correct weighting is obtained for light sources which vary rapidly across the face of a primitive. Normalization operations are excluded from the set of approximation sites.

Automatic handling of vector normalization can not be achieved using only the structural data types (such as float tuples) provided by current GPU languages. Instead, it is necessary to have distinct types for different vector quantities. In our type system we distinguish between points, vectors, unit vectors, and directions.

User-defined rules Our system also allows the user to specify user-defined rules regarding how frequencies should be assigned. These rules state whether, for a particular function (or overloaded version of a function), the algorithm should prefer placing it at the vertex or pixel processor. Also, it is possible to indicate whether the function should be included or excluded from the set of approximation sites. An example of such a user-defined rule is

```
clamp.canBeApproximated = False
clamp.prefer = PIXEL
```

that instructs the compiler to exclude applications of the function clamp from the set of approximation sites. Also, this function should, if possible, be computed in the pixel processor since it is not length-preserving and will, if computed per-vertex, give incorrect weighting across a primitive.

4.3 Shader examples

Table 1 lists a few example shaders together with the number of approximations sites before and after pruning. These examples span a number of commonly used shader techniques such as normal mapping, bump-mapping, environment mapping, and vertex deformations. In all these examples, texture accesses are restricted to only being allowed at per-pixel. This is the default behavior of our compiler, but it can be overridden on a per-texture or global basis, to allow vertex texture accesses, for instance.

Analyzing the results we see that shaders which depend on texture look-ups generally have fewer approximations sites than shaders relying on vertex data. For instance, variant I of the Blinn-Phong algorithm uses vertex normals and has 5 approximations sites. Variant II uses a normal map and has only a single approximation site representing the computation of the half-vector. The reason for this is that variant II must read the normal at the pixel level and, consequently, cannot perform lighting computations relying on the value of the normal at a lower level. The same holds for variant III. The shadow volume extrusion algorithm is a vertex-only algorithm and consequently has no approximation sites. Similarly, the ray-marching volume renderer does most

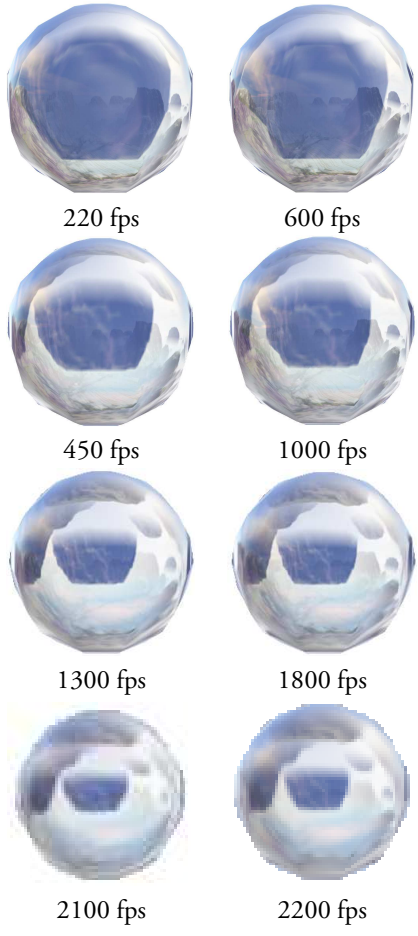


Figure 1: Sphere rendered using simulated glass at successively higher distances (top to bottom). The rendered results are scaled for easy comparison. The original on-screen coverage varies between 1.2M pixels and 1.2k pixels. The left column shows performance and visual results of using full pixel computations, whereas the right column uses vertex computations to the greatest extent possible. Using more vertex computations at medium distances can result in dramatic performance gains at a relatively small loss in visual quality. At extreme ranges the number of vertices approach the number of projected on-screen pixels making the two methods equally efficient. External control over where computations are performed enables such performance/visual quality "sweet spots" to be found quickly.

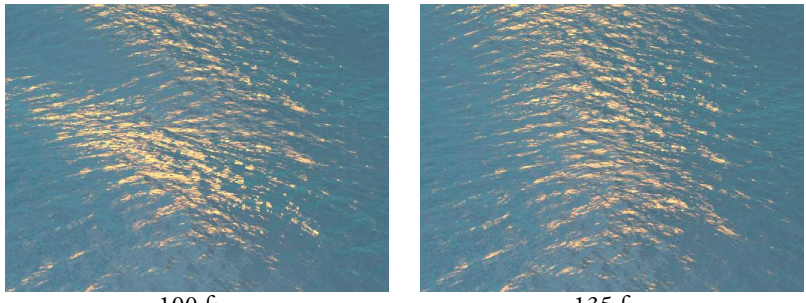
Shader	approx. sites	vertex instr.	pixel instr.
Blinn-Phong I	5 (11)	7–16	17–1
Blinn-Phong II	1 (3)	7-15	13–9
Blinn-Phong III	1 (4)	11–21	20–14
Shadow volume extr.	0 (0)	13	1
Volume ray marching	0 (2)	14	12
Refraction	4 (8)	17–34	22–4
Parallax mapping	1 (5)	16–28	22–13
Parallax mapping (LOD)	1 (5)	18–30	24–15
Animated water	9 (41)	24–43	77–59

Table 1: Example GPU programs together with the number of approximation sites and number of instructions in the resulting GPU vertex and pixel programs: three variants of Blinn-Phong shading using (I) per-vertex normals, (II) a model-space normal map, and (III) tangent-space normal perturbation using a bump map, respectively, GPU shadow volume extrusion, volume rendering using ray marching through a volume texture, simulated refraction using an environment map, parallax mapping (with and without LOD), and the water effect of Listing 1. The number of approximation sites before pruning are show in parenthesis.

of its work in the pixel stage marching through the volume texture. The vertex unit is used to compute clip-space positions and ray directions.

The refraction shader uses an environment map and simulated refraction and Fresnel terms to give a glass-like effect. Here, the whole or parts of the Fresnel approximation can be moved to the vertex shader. Parallax mapping simulates fine-scale surface features using a height-map to compute an offset in texture-space [13]. The LOD variant uses viewer distance to determine whether to apply parallax mapping or use simple texturing. The Blinn-Phong half-vector computation is the only approximation site in this case. In the water shader of Listing 1 the tangent and bi-normal are computed analytically from the wave equations and this can be done in either the vertex or pixel processing unit. The visual and performance difference of running the animated water and simulated refraction shaders using different distributions across the vertex and pixel processors are shown in Figures 1 and 2.

One important aspect to note is that the dependency analysis does not introduce any extra operations in the resulting shaders (apart from vector re-normalizations required for correct interpolation). In effect, the algorithm only chooses which values of a shader should be interpolated and passed to the pixel processing unit. Consequently performance of unified shaders is the same as that of handwritten shaders which have not been optimized using techniques outside the scope of this work, for example, lower precision data types, data-packing or texture function caching. The programmer maintains the same degree of performance control as in separate shader



100 fps 135 fps

Figure 2: Simulated water using Gerstner waves. The left variant uses exact per-pixel computations whereas the right uses the maximal amount of vertex computations, as inferred by our compiler. The visual effect of using the more correct, pixel-exact, variant can be seen clearly. Performance figures indicate the performance impact of using the more exact variant.

programming models of Cg/HLSL/GLSL.

5 Controlling approximations

Choosing which approximations should be applied to a particular shader is highly application and context dependent. For instance, moving the computation of the half-vector in Blinn-Phong shading from the pixel to the vertex shader is a valid approximation if both the viewer and light source are relatively far away. If this is not the case then this approximation will yield clearly visible artifacts. The approximation sites represent a convenient way of manipulating how parts of a program's computations should be distributed across the vertex and pixel processing units of the GPU.

We have implemented a basic artist's tool for quickly exploring the visual and performance impact of a given set of approximations. The tool lets the programmer or artist apply a unified GPU program to a model. The artist can then manipulate the shading algorithm by moving approximation sites between the vertex and pixel shader and getting direct visual feedback in a preview window. Additionally, numerical information such as peak signal-to-noise ratio, and max, average, and median pixel error, compared to the most accurate shader variant can optionally be displayed. This allows a shader to quickly be adapted to a particular mesh or application. The result of using this tool to generate shaders for discrete levels of detail (LODs) can be seen in Figure 1.

Using an algorithmic selection approach instead is another approach. With the pruned set of approximation sites, the Monte Carlo approach of Pellacini [24] or the

affine arithmetic approach of Heidrich et al. [11] can be used to select approximation sites with minimal visual impact. Note however, both these approaches are sensitive to the conditions of a particular application. For example, variable such as the distances of the observer and light sources can greatly affect the final visual result. Consequently, giving error bounds of the resulting shader is difficult without tight bounds on input parameters. This is therefore an area of research which is separate to our work. It is however complementary in the sense that different algorithms can easily be plugged in to our system.

6 Summary and discussion

We have shown that it is possible to write arbitrary GPU programs in a unified manner, without explicitly separating vertex and pixel computations. This includes support for current GPU features such as loops, conditionals, vertex texture fetches, etc. Our algorithm factors unified programs into vertex and pixel programs running on the GPU. By construction, the run-time performance of unified GPU programs is equal to that of programs written by hand using current high-level GPU languages. Unified shaders which only use vertex or pixel computations are automatically handled. For algorithms which require both vertex and pixel computations, the algorithm identifies computations that can be approximated by moving them from the pixel to the vertex processing stage. These approximation sites can be used by an artist to adapt and optimize a shader for a particular mesh or a particular set of application demands such as performance or hardware limitations.

Unified GPU programming enables truly high-level programming of the GPU where implementation details such as interpolation methods, vertex/pixel unit allocation, and CPU pre-computations are automatically handled by the compiler. This has several advantages. For example, modifying a shader to use a normal map instead of vertex normals is typically a two line change, whereas the same change using current GPU languages involves rewriting major parts of the vertex and pixel programs. Furthermore, since current restrictions and performance characteristics of the rendering pipeline are not explicitly encoded, the unified approach should be stable with respect to future architectural changes in the GPU.

An interesting future research direction is investigating how the unified GPU programming approach may be combined with the recently introduced *geometry shading units* [2]. Other interesting research areas, such as automatically compiling unified shaders to handle non-standard rendering techniques, image post-processing, or general purpose GPU computations, merits further investigation. Also, since our algorithm does not split loops or conditionals (see Section 4.2), adapting common compiler loop analysis techniques (such as dependency matrices) to GPU program frequency analysis is an area for future work.

```
uniform(environment, samplerCUBE)
uniform(bumpmap, sampler2D)
uniform(t, Float) # time
uniform(cameraPos, Point) # camera in model space
uniform(lightPos, Point) # light position in model space
uniform(deepColor, RGB, rgb(0,0,0.1))
uniform(shallowColor, RGB, rgb(0,0.5,0.5))
uniform(sunColor, RGB, rgb(1.0,0.5,0))

r0 = 0.02
ri = 1.0/1.33
waves = [(0.1, 3, 0.2, (0, −1)), (0.3, 1, 0.3, (−0.7, 0.7))]
bumpScales = [3,4]

# Helper functions
def gerstnerWave(A, Dx, Dz, f, p, k): # −> (dy, dydx, dydz)
    ...

def rescale(vec): return (vec − 0.5)∗2.0

def main():
    x,z = gl_Vertex.xz
    y, dydx, dydz = 0, 0, 0
    for f,A,p,(dx, dz) in waves:
        y, dydx, dydz += gerstnerWave(A, dx, dz, f, p, 2)
    pos = point(x,y,z)
    uv = gl_MultiTexCoord0.xy
    V = dir(cameraPos−pos)
    L = dir(lightPos − pos)
    N = normalize(vector(−dydx, 1, −dydz))
    T = normalize(vector(1, dydx, 0))
    B = normalize(vector(0, dydx, 1))
    for n in bumpScales:
        offset = rescale(tex2D(bumpmap, (2**n)∗uv).xyz)**(3−n)
        N += offset[0]∗T + offset[1]∗N + offset[2]∗B
    N = normalize(N)
    R = reflect(N, V)
    facing = 1−max(dot(V, N), 0)
    fresnel = r0 + (1−r0)∗pow(1−dot(V,N), 5)
    refl = texCUBE(environment, R)
    refr = texCUBE(environment, −refract(N, V, ri))
    specular = max(pow(dot(R,L), 100), 1)

    gl_Position = gl_ModelViewProjectionMatrix∗pos
    gl_FragColor = lerp(deepColor, shallowColor, facing) + \
                   lerp(refr.rgb, refl.rgb, fresnel) + \
                   sunColor∗specular
```

Listing 1: A single water shader program combining vertex deformations using Gerstner waves with per-pixel normal-perturbation using bump-mapping

Bibliography

[1] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 5, pages 286–292, 1978.

[2] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[3] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *Graphics hardware*, pages 51–59. Eurographics Association, 2003.

[4] Robert L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 11, pages 223–231. ACM Press, 1984.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, October 1991.

[6] Conal Elliott. Programming graphics processors functionally. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 45–56, 2004.

[7] Mark Finch. Effective water simulation from physical models. In Randima Fernando, editor, *GPU Gems*, chapter 1, pages pp. 5–29. Addison-Wesley, 2004.

[8] Alfred R. Fuller, Hari Krishnan, Karim Mahrous, Bernd Hamann, and Kenneth I. Joy. Real-time procedural volumetric fire. In *Symposium on Interactive 3D graphics and games*, pages 175–180, 2007.

[9] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[10] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 289–298, 1990.

[11] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Trans. Graph.*, 17(3):158–176, 1998.

[12] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4):196, 1996.

[13] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proceedings of the ICAT2001*, volume 12, pages 205–208, 2001.

[14] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Symposium on Interactive 3D graphics and games*, pages 39–46, 2007.

[15] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a GPU-based particle engine. In *Graphics hardware*, pages 115–122, 2004.

[16] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, 2003.

[17] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 149–158, 2001.

[18] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[19] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[20] Steven Molnar, John Eyles, and John Poulton. PixelFlow: high-speed rendering using image composition. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 231–240, 1992.

[21] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Graphics hardware*, pages 7–14. Eurographics Association, 2003.

[22] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: the PixelFlow shading system. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 25, pages 159–168, 1998.

[23] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 27, pages 425–432, 2000.

[24] Fabio Pellacini. User-configurable automatic shader simplification. *ACM Trans. Graph.*, 24(3):445–452, 2005.

[25] Ken Perlin. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 12, pages 287–296, 1985.

[26] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 159–170, 2001.

[27] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.

[28] Nick Thibieroz. Clever shader tricks. http://ati.amd.com/developer/SwedenTechDay/03_Clever_Shader_Tricks.pdf, October 2006.

[29] Steven Worley. A cellular texture basis function. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 23, pages 291–294, 1996.

Paper IV

Towards modular programming of the GPU pipeline

Calle Lejdfors and Lennart Ohlsson
Department of Computer Science
Lund University
Lund, Sweden

{calle.lejdfors|lennart.ohlsson}@cs.lth.se

ABSTRACT

Modern GPUs require separate programs to be written for the three different stages of the rasterization pipeline. This gives the programmer a high degree of control of the computations, but since the three programs are highly dependent on each other they are fragile and not easily changed or reused. In this paper we present a real-time shading language and compiler which allow the programmer to write geometry generation code and surface shading code in separate and composable modules. In neither of the two modules does the programmer specify a split between stages. This allocation is instead inferred and optimized by the compiler. For fine-tuning, an external interface makes it possible to tweak performance and visual quality through, for example, an interactive artist's tool. In experiments, the performance of our approach is equal to that of using hand-written optimized shaders.

Manuscript in preparation

1 Introduction

Programmable real-time graphics processing units (GPUs) have evolved tremendously since their introduction in 2001 [9]. Performance has increased by several orders of magnitude and the level of programmatic control advanced from only allowing simple *shaders* consisting of a few assembly instructions, to complex programs spanning hundreds of lines of high-level code. Furthermore, the original rasterization pipeline has been extended to support procedural geometry construction and manipulation [3].

A modern GPU consists of three programmable stages and to implement a shader, the programmer must write up to three separate programs that each implements a part of the shader. This gives great control over the GPU and enables a large number of complex effects to be implemented and combined.

However, these GPU programs are highly dependent on each other and typically contain code related to both geometry processing and shading. This makes GPU programs very fragile since changing one program (for example, by changing from per-vertex normals to normals maps) typically requires rewriting the other programs as well. Also, since geometry processing and shading are entangled, reusing GPU programs is difficult. This quickly leads to applications containing very many shaders that are only small variations of each other. This needless repetition is a problem in today's graphics applications [12, 2].

To handle this complexity, several approaches have been proposed [1, 11, 12]. By allowing shaders to be composed from existing shader fragments, they can be constructed and managed in a more modular fashion. As demonstrated by McGuire et al. [12], this makes developing shaders faster and easier, even to the point of allowing non-programmers to construct new shaders. However, these approaches are restricted to programming single stages in isolation, requiring the programmer to explicitly split the computations across the GPU. Also, none of these approaches supports procedural geometry construction.

In this paper we present a first step towards applying the modular approach to the full modern GPU pipeline. By extending the work on unified shaders [14, 7], we present a real-time shading language and compiler that separates geometry generation from surface shading. Both the geometry generator and surface shader are written as single programs with no explicit separation into GPU processing stages. Our compiler then combines a generator with a surface shader to efficient GPU programs. The main contributions made in this paper are:

- a compiler that enables modular programming of the modern GPU pipeline by allowing geometry generation to be separated from shading.
- an optimization algorithm that can optimize a combined geometry generator and surface shader across all stages of the GPU pipeline.

Furthermore, our code generation can be fine-tuned to control the exact performance

Figure 1: An overview of the pipeline of a modern GPU. White boxes represent the user-programmable stages and grey boxes represent fixed functionality.

and visual quality of the generated GPU code. This enables the programmer to use knowledge of the target application or hardware platform to optimize shaders just as when writing GPU programs by hand. We demonstrate how this ability can be used efficiently through a graphical user interface. In experiments, the GPU programs generated by our compiler are shown to be as efficient as hand-written code.

This paper is organized as follows; In Section 2 we give an introduction to modern GPU programming and previous work. In Section 3 we introduce our approach and in Section 4 we describe our analysis algorithm. In Section 5 we describe a number of examples with associated performance figures. Last, in Section 6 we conclude our paper with a discussion and some future work.

2 GPU programming

The current programming model for the GPU, exposed by programming interface such as OpenGL [16] and DirectX [3], closely reflects the actual hardware structure of the rasterization pipeline. Input vertex data is processed by the *vertex shader* after which it is combined into primitives that are passed to *geometry shader*. This stage uses the input primitive to output zero or more output primitives which are then clipped, rasterized, and finally passed to the *pixel shader* that computes the final per-pixel color. For a schematic overview of the rasterization pipeline see Figure 1.

A programmer controls this pipeline by writing programs that execute at the different programmable pipeline stages. These programs are written in either assembly languages, or more commonly, in dedicated C-like shading languages (e.g. GLSL [15], Cg [10], or HLSL [5]) that provide specialized data-types and operations that can be translated to efficient GPU code.

To simplify the construction of complex GPU shaders two major approaches have been proposed. The unified approach [14, 7] allows shaders to be written as single programs (just as in off-line systems such as RenderMan [6]), without explicitly splitting them across the programmable stages of the GPU. Instead, that is handled automatically by the compiler. The unified programming model simplifies shader programming and reduces fragility since the programmer need not explicitly move parts of a shader between the processing stages of the GPU, for example. However, support for modular development is limited to simple, procedural functions. Also, support for modern GPU features such as the geometry processing unit is lacking.

The second approach is a “building block” model [1] of programming, where

existing shader fragments are combined to construct new shaders. This encourages modular development and facilitates rapid construction and evaluation of shaders. For example, the *shader algebra* of McCool et al. [11] lets shaders be constructed using template meta-programming in C++. The *abstract shade trees* [12] allow designers to easily create effects by connecting primitives such as cube mapping and modulation using a GUI tool. However, both these approaches are limited to constructing shaders for a single pipeline stage and does not perform any analysis to exploit that parts of a shader can be moved to an earlier pipeline stage. This is an important optimization that can, for example, reduce computational pressure on the pixel processing unit significantly. Furthermore, none of these approaches support geometry shaders.

3 Modular unified GPU programming

The development model provided by the building block approach makes developing shaders faster and easier. However, the technique is restricted to programming only single pipeline stages in isolation. This makes it difficult to extend to handle the GPU's ability to dynamically construct and manipulate geometry (where simultaneous control over multiple stages is required). The unified programming approach allows flexible programming of multiple stages, but the support for modular development is limited.

We argue that the benefits of both methods can be combined by separating the responsibilities of the GPU into two distinct areas: *geometry generation* and *surface shading*. These areas can then be programmed separately, allowing GPU shaders to be constructed by combining different geometry processing algorithms with different shading algorithms. Both the geometry generators and surface shaders are specified as single programs, with no explicit separation into different pipeline stages. Then, using inter-stage optimization, the resulting GPU programs can be made very efficient. Furthermore, by allowing the user to interactively guide the code generator, performance equal to that of hand-written shaders is achieved.

In this paper we present a compiler and shading language that implements this modular unified GPU programming model. Our system lets the programmer write separate geometry generators and surface shaders that can be combined to form efficient GPU shader programs.

3.1 An embedded GPU language

We have implemented our shader programming approach as a embedded language [4] in Python, an object-oriented, dynamically typed, language. Using language embedding lets us avoid the added work of implementing a complete language front-end and enables us to quickly experiment with different features and implementation approaches.

```
uniform(Float, phi)
uniform(Float, width)
varying(UnitVector, ViewSpace, viewN)
varying(Point, ViewSpace, viewPos)

def geometry():
    Ps = gl_ModelViewMatrix*gl_PositionIn
    Ns = normalize(gl_NormalMatrix*gl_NormalIn)
    T = direction(Ps[1] - Ps[0])
    Bs = cross(T, Ns)
    offsets = width*(cos(phi)*Bs + sin(phi)*Ns)
    for i in xrange(2):
        viewPos, viewN = Ps[i] - offsets[i], Ns[i]
        gl_Position = gl_ProjectionMatrix*viewPos
        EmitVertex()

        viewPos, viewN = Ps[i] + offsets[i], Ns[i]
        gl_Position = gl_ProjectionMatrix*viewPos
        EmitVertex()
```

Listing 1: A geometry generator that uses a line with additional normal information to construct a ribbon. The twist of the ribbon can be controlled by the phi parameter. This can be combined with a surface shader to obtain a full GPU shader.

The shading language is based on GLSL [15] but provides a richer type system. The compiler uses forward type inference to track and infer the types of expressions in the shader. We use a type system similar to the *semantic types* of McGuire et al. [12] that enables the compiler to also track the space and intended usage of a vector quantity. Semantic types encode that, for example, a vector is expressed in model space and that it is used to encode a direction. This information can then be used to improve error detection (e.g. by catching inconsistent space usage) and to correctly interpolate vector data.

Geometry generators are written just as geometry shader programs where no computations have been performed in the vertex program; they accept input primitives expressed in model space and outputs zero or more primitives (possibly of another type) in clip space. These primitives can be either points, lines, or triangles, and may optionally contain adjacency information (adjacent line segments, or triangles that share edges with the input triangle). The input data is provided in the arrays gl_PositionIn, gl_NormalIn, etc., named like their GLSL counterparts. Output data that should be interpolated across the output primitives are declared *varying*. User-defined varying data and the clip-space position must be written for each output vertex before it is emitted. Parameters that do not vary across the mesh, such as light positions or textures, are declared *uniform*.

Shaders are written like GLSL pixel shader programs; they accept varying data

```
uniform(Point, ViewSpace, lightPos)
varying(UnitVector, ViewSpace, viewN)
varying(Point, ViewSpace, viewPos)

def surface():
    L = direction(lightPos − viewPos)
    V = direction(−viewPos);
    H = direction(L+V)
    diffuse = clamp(dot(viewN, L), 0.0, 1.0)
    specular = clamp(pow(dot(viewN, H), 32.0), 0.0, 1.0)
    gl_FragColor = diffuse∗rgb(1,0,0) + specular∗rgb(1,1,1)
```

Listing 2: A Blinn-Phong surface shader implemented in our language. This shader accepts surface positions and normal in view space and computes the incident light in each pixel. The final pixel color is, just as in GLSL, written to the dedicated *gl_FragColor* output variable.

provided by the geometry generator and, together with uniform information, computes the final per-pixel color that is written to the dedicated gl_FragColor variable. The shader can use the varying output data computed by the generator to compute the per-pixel color of a mesh. To be able to combine a shader with a geometry generator, the varying data computed by the generator must match the varying inputs required by the shader. Here, semantic types are used to ensure that varying data is of the correct type and is expressed in the appropriate space.

For an example of a geometry generator, consider Listing 1. This generator uses a line strip with additional normal information to construct a twisting ribbon along that line. This can be used, for example, in a special effects system to dynamically construct streamers or trailers that follows an object. The generator constructs a local frame from the direction of the line and the normal. This frame is then used to construct a triangle strip representing the tessellated ribbon. To animate the ribbon, we twist it around itself by the angle phi. This generator computes the view space position and normal for each vertex (declared as varying) which can be used by the surface shader to compute the final per-pixel color. An example Blinn-Phong surface shader is shown Listing 2. The result of combining the ribbon generator with the Blinn-Phong surface shader is shown in Figure 2.

To allow transformations to be applied in parallel to several objects, our language provides array programming facilities [13]. This enables, for example, in the ribbon generator above, both input vertices of the line segment to be transformed to view space in a single line. Similarly, the normals, and offset vectors are computed using array operations. Also, just as the host language Python, our embedded language supports parallel assignment (e.g. a,b = b,a). Together, these features provide a convenient and concise syntax. Using these features incurs no extra overhead in the resulting GPU programs, as all such operations are inlined by the compiler.

4 Operation scheduling and analysis

The task of the compiler is to analyze the combination of a geometry generator and a surface shader to determine at which GPU processing stage each operation should be scheduled. This should done in a way that gives the best performance possible without introducing numerical errors. In general, it is more efficient to do computations earlier in the rendering pipeline. This heuristic is based on the fact that the number of on-screen pixels usually far greater that the number of vertices created by the geometry shader. Similarly, for geometry shaders, using the vertex shader is typically more efficient since vertices are often shared between multiple primitives. Also the vertex shader can be more efficiently executed in parallel [3].

This means that for surface shaders, operations should, if possible, be scheduled to the geometry or vertex program. To achieve this, each term is analysed to determine if it varies linearly over primitives, i.e. if it can be computed per-vertex and then linearly interpolated without loss of precision. If it does then it can be computed per-vertex, otherwise it must be computed in the pixel program [7]. To achieve correct results when computing vector data per-vertex, the interpolation method is chosen according to the semantic type of the vector.

For geometry generators, our compiler first combines the per-vertex part of the surface shader with the geometry generator. The result is then analyzed to find which parts may be computed in the vertex shader and which parts must be scheduled to the geometry shader. To do this our compiler recursively, breadth-first, checks how each attribute of each vertex of the input primitive is transformed. Terms that are computed from input attributes in the same way for every input vertex are scheduled in the vertex program. Terms that are not computed in the same way for all vertices, or which depend on data from more than one vertex, are put in the geometry program.

For example, consider the following generator fragment that computes the normal of an input triangle in view space:

```
p1 = gl_ModelViewMatrix*gl_PositionIn[0]
p2 = gl_ModelViewMatrix*gl_PositionIn[1]
p3 = gl_ModelViewMatrix*gl_PositionIn[2]
N = cross(p3-p1, p2-p1)
```

Here, the position of each input vertex is transformed to view space. Our algorithm will start at the computation of `p1` and then check the other uses of `gl_PositionIn`. It will then find that `p2` and `p3` are computed using the same sequence of operations as `p1`. Hence the model to view space transformation can be performed in the vertex program. The computation of the normal `N` must be performed in the geometry program since it depends on data from more than one vertex.

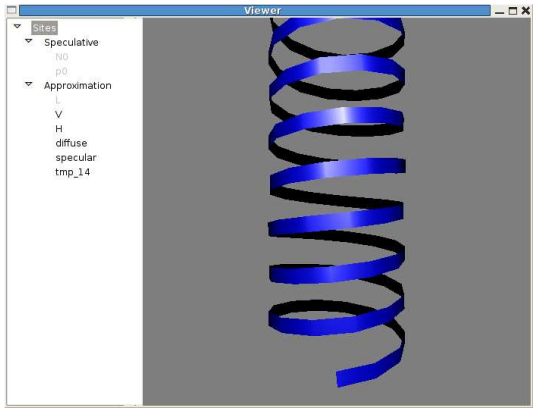


Figure 2: A graphical user interface for controlling the placement of speculative and approximative terms of a generator and surface shader pair. Currently, the ribbon generator combined with the Blinn-phong shader of Listings 1 and 2 is shown. The left pane shows the speculative and approximative terms that can be manipulated. The right pane shows a preview of the current generator and surface shader. Information about performance and the number of instructions in the vertex, geometry, and pixel GPU shaders can optionally be displayed.

4.1 Performance optimization

The heuristic above, that operations should be scheduled as early as possible in the pipeline, works for most examples (see Section 5). However, there are some cases where it is difficult to statically estimate the performance of a shader. As an example of such a case, consider this implementation of a back-face culling geometry generator:

```
p1,p2,p3 = gl_ModelViewMatrix*gl_PositionIn
N = cross(p3-p1, p2-p1)
if dot(p1, N) <= 0:
    for i in range(3):
        gl_Position = gl_ProjectionMatrix*vs[i]
        EmitVertex()
```

Here, our algorithm will find that the transformation to view space should be performed per-vertex. Also, it will determine that the projection could be executed in the vertex program, but since the conditional must be executed in the geometry program, so will the projection. In our test scene, the above shader runs at 149 frames per second (FPS). But, if the projection is *speculatively* computed in the vertex program (even though this value will not be needed for all vertices) it runs at 241 FPS, an improvement of more than 60 percent.

To be able to handle cases where the heuristic results in sub-optimal performance, our code generator can be controlled using an external application programming in-

		Performance (frames per second)			
Geometry generator	Surf. shader	Hand-w.	Naive	Our appr.	w/ opt.
Ribbon	Blinn-Phong	13.2	11.8	12.0	13.2
Motion blurred PS	Texturing	20.0	19.5	20.0	20.0
Marching cubes	Blinn-Phong	44.3	29.3	44.3	44.3
Adaptive tessellation	Blinn-Phong	15.5	15.5	13.7	15.5
Fin generator	Texturing	7.0	5.5	7.0	7.0
Back-face culling	Constant color	241	121	149	241

Table 1: Performance results of a number of combinations of geometry generators and surface shaders. The naive translation executes the entire geometry generator in the geometry program, and the surface shader in the pixel program. Our approach uses the optimization algorithm to schedule parts of the shader earlier in the pipeline. The optimized variant has been optimized using the external control mechanism. The hand-written variants are manually optimized to provide the maximal overall performance for the respective test scenes.

terface (API). Using this interface it is possible to move parts of generator between the vertex and geometry pipeline stages. Also, it allows parts of the surface shader to be *approximated* by performing computation per-vertex rather than per-pixel [7]. This typically results in better performance at the expense of visual quality (since a non-linear function is approximated by a piece-wise linear one).

This kind of fine-grained control enables the programmer to exploit hard-to-quantify knowledge about the application (such as expected number of branches takes for a particular mesh, mesh resolution w.r.t. expected viewing distance, etc.) and target hardware to adapt the shader just as when writing shaders by hand. Using our approach, this kind of manipulation can be performed quickly, without rewriting any parts of the shader.

To demonstrate how the external interface can be used to optimize a geometry generator and surface shader, we have implemented a graphical user interface (GUI) that lets the scheduling of speculative or approximative terms be controlled. The visual quality and performance change resulting from moving a term is shown in real-time allowing a performance/visual quality sweet-spot to be located quickly. The GUI is shown in Figure 2.

5 Examples

To evaluate our modular programming approach, we have tested our compiler on a number of different geometry generators and surface shaders. Both generators and shaders are developed separately. Table 1 shows some combinations on which we have applied our compiler.

The ribbon and back-face culling generators are described above (Sections 3.1 and 4.1, respectively). The motion blurred particle system attempts to simulate the streaking effect caused by rapidly moving objects. It uses a geometry generator to construct one or more transparent daughter particles along the path travelled by the original particle during the last frame. When blended into the framebuffer, this gives the appearance of a quickly moving particle. The transparency of each spawned particle is available as varying input to the surface shader that can use it to draw each particle with the appropriate alpha value.

The marching cubes generator extracts a iso-surface from a volume data set. The normal used by the surface shader is constructed from each generated triangle facet. Note that this generator can not be combined with a texturing shader since it can not compute meaningful 2d texture coordinates. The adaptive tessellation example accepts a point input primitives and dynamically creates a sphere with a tessellation level dependent on the viewer distance. Last, the fin geometry generator constructs "fins," used when rendering fur [8], orthogonal to the mesh's silhouette.

5.1 Performance comparison

For each pair of generator and surface shader we have compared the performance of four different versions of the same shader (see Table 1). In all examples the performance of the optimized versions of our approach is equal to that of the hand-written versions. Also, in all but one example our compiler either outperforms or give performance equal to the naive translation. This demonstrates how code hoisting, where possible, gives a solid speed-up by exploiting the vertex processing unit.

In the adaptive tessellation example, the our compiler actually performs worse than the naive translation, just like a programmer probably would. The reason is that the compiler, just like the programmer, will schedule parts of the surface shader to the geometry program. In this case the overall performance is lower than it would have been, had the shading been performed entirely in the pixel program. Presumably this is a result of the geometry shader being slower, and the need to interpolate an extra varying vector for each triangle. Typically this kind of effect are hard to predict, even for an experienced programmer.

Together with the back-face culling example of Section 4.1, the adaptive tessellation shows the difficulty of trying predict the performance of a GPU shader that uses the geometry processing unit. These examples demonstrate the benefit of our external interface, which enables finding the optimally performing variants quicker than rewriting source code.

6 Discussion

We have presented a language, and a compiler and analysis framework that enables modular programming of the rasterization pipeline of a modern GPU. It enables a geometry generator to be combined with a surface shader and the result is translated to efficient code running on the GPU. We have shown that for a number of typical uses of the geometry shading hardware, the code generated by our compiler performs equal to that of hand-written, manually optimized GPU programs. We also provide an interface that enables the code generator to be controlled. This lets the programmer tune and adapt the generated programs to specific hardware or application requirements.

The performance results in this paper highlight the difficulty of estimating the performance of a shader that uses the geometry shading unit. Here, the ability of our approach to control the code generation, enabling computations to be quickly moved between the functional units of the GPU, considerably simplifies and speeds up development and optimization. A natural future extension would be to include the programmable tessellation units [17] available on AMD/ATI hardware.

Bibliography

[1] Gregory D. Abram and Turner Whitted. Building block shaders. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 283–288, 1990.

[2] Johan Andersson and Natalya Tatarchuk. Frostbite rendering architecture and real-time procedural shading & texturing techniques. AMD Sponsored Session. GDC 2007, March 2007.

[3] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[4] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27. Springer-Verlag, 2000.

[5] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[6] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 289–298, 1990.

[7] Calle Lejdfors and Lennart Ohlsson. Unified GPU programming. Technical Report LU-CS-TR:2008-244, Department of Computer Science, Lund University, 2008.

[8] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *I3D '01*, pages 227–232, New York, NY, USA, 2001. ACM.

[9] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 149–158, 2001.

[10] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[11] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[12] Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract shade trees. In *I3D '06*, pages 79–86, New York, NY, USA, 2006. ACM.

[13] Philippe Mougin and Stéphane Ducasse. Oopal: integrating array programming in object-oriented programming. In *OOPSLA '03*, pages 65–77, New York, NY, USA, 2003. ACM.

[14] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 159–170, 2001.

[15] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.

[16] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide*. Addison-Wesley Proffesional, 5th edition, 2005.

[17] Natalaya Tatarchuk. Real-time tessellation on the GPU. SIGGRAPH course, August 2007.

Paper V

Space-free shader programming: Automatic space inference and optimization for real-time shaders

Calle Lejdfors and Lennart Ohlsson
Department of Computer Science
Lund University
Lund, Sweden

{calle.lejdfors|lennart.ohlsson}@cs.lth.se

ABSTRACT

The graphics processing units (GPUs) used in todays personal computers can be programmed to compute the visual appearance of three-dimensional objects in real time. Such programs are called shaders and are written in high-level domain-specific languages that can produce very efficient programs. However, to exploit this efficiency, the programmer must explicitly express space transformations necessary to implement a computation. Unfortunately, this makes programming GPUs more error prone and reduces portability of the shader programs.

In this paper we show that these explicit transformations can be removed without sacrificing performance. Instead we can automatically infer the set of transformations necessary to implement the shader in the same way as an experienced programmer would. This enables shaders to be written in a cleaner, more portable, manner and to be more readily reused. Furthermore, errors resulting from incorrect transformation usage or space assumptions are eliminated. In the paper we present an inferencing algorithm as well as a prototype space-free shading language implemented as an embedded language in Python.

Accepted for publication in *Proceedings of TPCG '08*, 2008.

1 Introduction

The *graphics processing unit* (GPU) available on modern graphics cards makes it possible to execute user-defined *shader programs* in real-time. Using these *vertex* and *pixel shaders* it is possible to control the position and orientation, as well as the per-pixel color of rendered objects. Programmable shading hardware enables many highly realistic graphical effects to be implemented.

Shaders typically makes heavy use of vector calculations to compute the appearance of a mesh. When implement them using current GPU languages [5, 2, 10], it is necessary to perform several explicit coordinate or *space* transformations to a common space in which the appearance of the mesh may be computed. By using that, in general, there are more projected on-screen pixels than there are vertices in a mesh, it is possible to get very efficient shaders by performing the majority of these transformations per-vertex. By choosing an appropriate space, it is even possible to avoid some transformations completely.

The need to perform explicit transformations put an additional burden on the shader programmer. Moreover, the choice of space is dependent both on the application and the model that is to be rendered. This can result in a shader performing suboptimally, or even incorrectly, when used in another application or for another model. This type of implicit assumptions together with the high performance-sensitivity of shaders, results in very tight coupling between shader and application. This makes reusing shaders in other applications, or even for other models, difficult.

In this paper we show that explicit transforms are not required. Instead, it is possible to automatically infer the same choice of space transforms that an experienced programmer would make. The basis for our *space-free* shader programming approach is a type system that is richer than those provided by other GPU languages and an inferencing algorithm based on this type system. We have developed a prototype compiler that implements space-free shader programming. In this language shaders are written similar to off-line shaders, as though every vector and point is already in an appropriate space. By removing explicit transforms in this way we increase portability and reuse, since the code no longer contains application-specific assumptions. Also, consistent usage of vector quantities is guaranteed.

As an example, consider the Lambertian (diffuse) lighting model where the light reflected from a point is proportional to cosine of the angle between the surface normal and the direction of the light source, i.e. the dotproduct of these two unit vectors. Listing 1 shows this shader written in GLSL and Listing 2 shows the same shader implemented in our space free language. In both versions the `vertex` function is executed once for every vertex of the mesh, and the `pixel` function once for every projected on-screen pixel. Data that should be interpolated across the primitive and used as input to the pixel program is declared as `varying`. The vertex program writes values to the varying parameters (surface normal `N` and vector to the light source `L` in this example)

```
varying vec3 L;
varying vec3 N;

void vertex() {
   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
   N = normalize(gl_NormalMatrix*gl_Normal);
   L = (gl_LightSource[0].position −
                              gl_ModelViewMatrix*gl_Vertex).xyz;
}

void pixel() {
   vec3 Nn = normalize(N);
   vec3 Ln = normalize(L);
   float diffuse = max(dot(Nn, Ln), 0.0);
   gl_FragColor = diffuse*vec4(1,1,1,1);
}
```

Listing 1: A Lambertian (diffuse) shading algorithm implemented in GLSL using view space.

and the clip space position of the vertex (written to the dedicated variable gl_Position). The pixel program reads the interpolated result and uses it to compute the final pixel or fragment color which is written to the variable gl_FragColor.

The differences are that in the GLSL version, lighting is computed in view space and the necessary space transformations are performed explicitly by matrix multiplication. In the space-free version the choice of space and the necessary matrix multiplications are instead inferred and inserted by the compiler. Also, in the GLSL version, explicit re-normalization of unit vectors is required. In the space-free version this is automatically handled by the compiler.

In the rest of the paper we describe our prototype shading language and the implementation of our compiler.

2 Automatic space inference

Translating a space-free shader to the GPU consists of finding a set of spaces and transformations such that the generated program is both correct and gives the highest possible run-time performance. Here, correctness means that the result of the shading algorithm must be identical to the result which would have been obtained using a known, correct choice of spaces. This involves ensuring that each operation in the shader is evaluated in an appropriate space. For example, consider computing the dot-product dot(N,L) used in the Lambertian shader in Listing 1. By default, this quantity should be computed in view space, since we are interested in computing how a mesh appears to the viewer.

```
varying(unitvector, L)
varying(direction, N)

def vertex():
    gl_Position = gl_Vertex
    N = gl_Normal
    L = gl_LightSource[0].position - gl_Vertex

def pixel():
    diffuse = max(dot(N, L), 0.0)
    gl_FragColor = diffuse*rgba(1,1,1,1)
```

Listing 2: A Lambertian shading algorithm in our prototype language. Here no explicit vector transforms are required. Also, explicit re-normalization of L in the pixel shader is automatically performed by the compiler.

However, suppose we know that model space and view space are related via a unitary transform U, i.e. a transform that is both angle and length-preserving. Then, using the identity for unitary transforms

$$\mathsf{dot}(\mathsf{N}_v, \mathsf{L}_v) = \mathsf{dot}(\mathsf{U}*\mathsf{N}_m, \mathsf{U}*\mathsf{L}_m) = \mathsf{dot}(\mathsf{N}_m, \mathsf{L}_m)$$

we find that we may compute the scalar product in model space without introducing errors. Here the *m* and *v* subscripts denote the vector expressed in the view and model space frames, respectively. So, if N and L are expressed in model space, it is possible to avoid two transforms in this case.

In order to deduce which spaces are possible for a particular computation we must first, know what spaces and transforms are available, and how these transforms behave. Second, we must determine how each operation behaves under these space transformations. And, third, we must know the type of every expression in the shader to know how it should be transformed (e.g. unit vector should be transformed to unit vectors, normals to normals, and so on). Using this information, we may then deduce every possible space in which the particular shader can be implemented. By conservatively inferring the possible spaces for each operation the resulting shader can be guaranteed to be correct from a space usage perspective. Then, analyzing the cost of the different choices of transforms, we can choose the variant with the highest run-time performance.

2.1 Spaces and transforms

The spaces and transforms available are in most cases determined by the application. The application is responsible for placing and orienting the viewer in the world (thus determining the view-to-world and world-to-view transforms) and also, for placing each model in world (hence giving the model-to-world transform for each model).

Some applications do not represent world space explicitly, instead preferring to represent the model-to-view transforms directly.

Furthermore, some meshes contain information about local spaces, such as *tangent space*, used in more complicated shading algorithms (see Section 4). In this case, these spaces are constructed explicitly in the shader and are available to the space inference algorithm just as any other space.

Operations can be divided into three classes, depending on how they behave under space transforms:

- space independent – does not change with changing space. Includes all scalar operations (such as sine, cosine, exp, max, min etc.) and color operations.
- space dependent – operations which require arguments to be in the same space (vector addition, subtraction, ...) but does not involve angle or length measurements.
- metric dependent – operations which require arguments to be in the same space and have the same metric (dot-product, normalization, length, etc.).

This classification lets us determine, given that an operations should be computed in a particular space, the set of possible spaces for that operation.

Space independent For a space independent operation that should be computed in a space S, the only possible space is S. For example, computing the final pixel color by multiplying a color by a floating point value

$$\mathsf{gl\_FragColor} = \mathsf{diffuse} * \mathsf{materialColor}$$

should be performed in view space. This follows by the reasoning above, that the pixel color should be computed as it appears to the viewer.

Space dependent For a *space dependent* operation that should be computed in a space S the set of possible spaces is equal to those spaces from which there exists a transform to S. For example, suppose we should compute the light vector L in view space by

$$\mathsf{L} = \mathsf{lightPos} - \mathsf{vertexPos}$$

Then we may compute the light position and vertex position in any space for which there exist a transform to view space. The resulting vector can then be transformed to view space using this transform. So, if there exist a transform from model to view space and we have the light and vertex position in model space. Then, we can compute the light vector in view space by

$$\mathsf{L} = \mathsf{modelToView} * (\mathsf{lightPos} - \mathsf{vertexPos})$$

where modelToView is the transformation matrix from view to model space. This saves us one transform.

Metric dependent In the previous section we saw an example of *metric dependence* when we argued that we could compute the diffuse term in model space rather than view space, when these spaces were related by a unitary transform. Generally, if an operation is metric dependent and should be computed in some space S then it may be computed in any space from which there exists a unitary transform to S.

It is possible to generalize metric dependence further by considering transforms which are angle but not length-preserving. For such a transform T we have

$$\mathsf{dot}(\mathsf{T}*\mathsf{v},\ \mathsf{T}*\mathsf{u}) = \lambda^2 * \mathsf{dot}(\mathsf{v},\ \mathsf{u})$$

for arbitrary vectors v and u and where $\lambda = \det \mathsf{T}$. However, due to lack of motivating real-world examples, we restrict ourselves to the unitary case, i.e. when $\lambda = 1$.

2.2 Typing and vector transforms

How a vector quantity is transformed is dependent on what type of vector it is and what kind of information it encodes. The type of the transformed vector must be the same as the original vector's. For example, transforming a unit vector **v** using a transform T must result in a new unit vector. Hence, this transform must be done by

$$\mathsf{v}' = \mathsf{T}*\mathsf{v}/\mathsf{length}(\mathsf{T}*\mathsf{v})$$

Here we can use transform properties to simplify this expression. For example, if we know that T is unitary, then $\mathsf{length}(\mathsf{T}*\mathsf{v}) = 1$ and the division can be skipped. However, if the vector v is a surface normal, then it should be transformed using the inverse transpose of T instead. This is necessary to ensure that the result is still a normal, i.e. orthogonal to the surface.

These examples illustrate the need for a fine-grained type system to be able to deduce how a vector quantity should be transformed. We use a type system which is an extensible form of the *semantic types* of McGuire et al. [6]. This type system provides types for vectors, points, unit vectors, directions, and normals, in addition to the vector types provided by GLSL. Similarly, the matrix types of GLSL are extended to include 4×4 unitary and 3×3 unitary matrices.

In addition to being used to deduce how vectors should be transformed, type information can be used to correctly interpolate values between the vertex and pixel shader. Two methods are interpolation methods are used: unit vectors are normalized first in the vertex shader, interpolated, and then re-normalized in the pixel shader. Directions, on the other hand, must not be normalized in the vertex shader, only in the pixel shader. Our system uses type information to chose the correct method depending on the type of varying data. In current shader languages, this must be handled manually by the shader programmer and it is a frequent source of errors.

2.3 Shader cost optimization

Selecting the best space choice can be done by estimating the run-time cost of every shader variant, and choosing the variant having the lowest cost. However, since every shader contains code executing per-vertex and per-pixel, the *computational frequency* [9] at which each operation of the shader executes must be taken into consideration when choosing the best variant. These frequencies are:

- Per-model or instance computations such as transform matrices and light positions which do not vary across the mesh. Computed on the CPU and then downloaded to the GPU.
- Per-vertex calculations for computing primitive interpolants and attributes such as texture coordinates, shadow map coordinates, and, depending in lighting model, light directions and tangent space transforms. Performed in the vertex processing unit of the GPU.
- Per-pixel calculations including texture accesses and possible pixel discards performed in the pixel processing unit of the GPU. This step is performed before alpha, stencil, and depth testing.

It is not possible to *a priori* determine how many times each part of a shader will be executed. However, for the majority of applications, the general rule is that the pixel shader is executed many more times than the vertex shader. This follows since there generally are many more projected on-screen pixels than vertices of a mesh. Similarly, the vertex shader is executed more often than the model computations, as model computations are performed only once for each mesh which typically consist of several thousand vertices.

This asymmetry can be described by summing the run-time cost of each individual step and representing them by a tuple (t_p, t_v, t_m) for the pixel, vertex, and model run-time cost, respectively. These tuples can then be compared using lexicographic ordering to find a least element. By using another ordering function it is possible to adapt the choice algorithm to optimize for non-standard applications, such as wireframe renderers (where the ratio of pixels to vertices is lower).

2.4 Further optimizations

When constructing shader variants there are two important optimizations that need to be considered to correctly determine the cost of a shader. The first is that unitary transforms can be inverted without cost. Using that the GPU does not distinguish between row and column vectors we may perform the following rewrite:

$$\mathsf{T}^{-1} * \mathsf{v} = \mathsf{T}^{t} * \mathsf{v} = (\mathsf{v} * \mathsf{T})^{t} = \mathsf{v} * \mathsf{T}$$

using the fact that the inverse of a unitary transform is its transpose. This optimization is particularly important when transforming surface normals since such vectors should be transformed using the inverse transpose of the matrix used to transform the surface. Hence, for a unitary transform U we get $(\mathsf{U}^{-1})^t = (\mathsf{U}^t)^t = \mathsf{U}$.

Second, when constructing local frames in the shader, the vectors used to defined the coordinate system for the space have constant coordinates in that space. For example, if a shader constructs a local space S from the basis vectors **u**, **v**, and **w**. Then we may rewrite, for example:

$$\mathsf{dot}(\mathsf{u},\ \mathsf{l}) = \mathsf{l.x}$$

i.e. as the first component of vector **l**, if the dot product is computed in the space S. This is possible since in S the coordinates for **u**, **v**, and **w** are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively. Also, if these vectors are used as interpolants, then we can further reduce computational requirements by not interpolating them, since they are constant in S.

3 Implementation

Our prototype, space-free shading language is implemented as an embedded language in Python, an object-oriented, interpreted, dynamically typed, high-level language. Language embedding allows us to reuse parts of the Python compiler framework to avoid writing a complete language front-end. This has enabled us to quickly experiment with different features and design choices in our implementation. For an example Lambertian shader, see Listing 2.

Just as in GLSL, our language provide a number of implicit input and output parameters (vertex position, pixel color, texture coordinates) named after their GLSL counterparts. These parameters are prefixed by gl_ and are provided either in model or view space. A non-exhaustive list of implicit parameters along with their type and space is given in Table 1.

3.1 The space inference process

Our space inference compiler is implemented as a constraint solving process. Starting at the output variables gl_Position and gl_FragColor for which the desired output spaces are known, it proceeds breadth-first back through the data-dependence graph of the program. For every operation, a space is chosen from the set of possible space (using the classification in Section 2.1) and this choice is propagated as a constraint back through the data-dependence graph. Once spaces have been chosen for every operation, the shader variant is passed to a backend code generator.

The backend inserts the necessary transforms to implement the shader in the chosen spaces and then optimizes the shader (in particular using the domain specific

Parameter	Type	Space	Notes
gl_Vertex	Point	ModelSpace	
gl_Normal	Normal	ModelSpace	
gl_Position	Point	ClipSpace	Must write per-vertex
gl_FragColor	RGBA	ViewSpace	Must write per-pixel
gl_LightSource[i]			
.position	Point	ViewSpace	

Table 1: A non-exhaustive list of implicit variables provided by our prototype language. The output variables gl_Position and gl_FragColor must, as indicated, be computed and written in the vertex and pixel shader, respectively.

optimization in Section 2.4). After this a frequency analysis is performed, any computations which do not vary per-vertex or per-pixel are moved to the CPU to avoid unnecessary re-computations. Also, as part of this analysis, the shader cost at the model, vertex, and pixel frequency are estimated. Finally the GPU vertex and pixel shader programs are generated. The code generator currently targets Python for the CPU code and GLSL for the GPU code.

3.2 Shader optimization

The space inference process is continued until the set of possible spaces has been exhaustively searched. Using the estimated shader cost computed by the backend the shader variant with the lowest run-time cost is found. Currently, the cost ranking is performed using lexicographic ordering as described in Section 2.3. The cost ranking function is passed as a parameter to the compiler allowing it to be replaced by a non-standard ranking method if required.

As it is currently implemented, the running time of our compiler is proportional to the number of possible space choices in a shader (worst case: exponential in the number of vector operations). Reducing this search time is not straightforward. Traditional methods for reducing the run-time for minimization algorithm relies on pruning or cutting down the potential search space. However, due to the potentially large performance impact of the optimizations in Section 2.4, the run-time cost of a shader is difficult to reliably estimate before the space for each operation has been chosen. These optimizations can lead to entire expression being eliminated or replaced by a simpler one. For example, if T is unitary and $\mathbf{v}$ is the first coordinate axis of T then

$$\mathsf{dot}((\mathsf{T}^{-1})^t{*}\mathsf{v},\ \mathsf{u}) = \mathsf{dot}(\mathsf{T}{*}\mathsf{v},\ \mathsf{u}) = \mathsf{dot}((1,0,0),\ \mathsf{u}) = \mathsf{u.x}$$

In this case a dot product computations and a transform has been reduced to a single vector member access (which can be performed without run-time cost on the GPU). Compilation times for our examples ranges from sub-second to at most a few seconds

```
uniform(sampler2D, bumpmap)
attribute(vector, ModelSpace, tangent)
varying(vector, L)
varying(vector, H)
constant(point, ViewSpace, eyePos, (0,0,0))

def vertex():
    gl_Position = gl_Vertex
    gl_TexCoord[0] = gl_MultiTexCoord0
    N = gl_Normal
    T = tangent
    B = cross(N, T)
    space(TangentSpace, unitary3(T,N,B))
    L = gl_LightSource[0].position - gl_Vertex
    V = eyePos - gl_Vertex
    H = normalize(L+V)

def pixel():
    offset = TangentSpace(tex2D(bumpmap,gl_TexCoord[0]).xyz)
    N = normalize(offset + N)
    diffuse = max(dot(N, L), 0.0)
    specular = pow(max(dot(N, H), 0.0), 8)
    gl_FragColor = diffuse*rgba(1,0,0,1)+specular*rgba(1,1,1,1)
```

Listing 3: A space-free bump mapping shader. The surface normal is perturbed using a texture to give the impression of fine-scale structure such as bumps or ridges on a surface. Here, the vertex shader is used to construct a tangent frame which can be used for shading computations. Note that the space of the offset vector and eye position must be explicitly specified for the inference algorithm to work.

and we have not found this to be a problem. Nevertheless, reducing compilation times is an interesting area for further research.

Another item of note is that space inference algorithm will never place a transform inside a loop. If a transform were to be placed inside a loop, the run-time cost of the generated shader can not be estimated since the number of loop iterations can in general not be determined. In some restricted cases, it is possible to statically determine the number of times the body of a loop will be executed. However, in non-trivial examples the number of loop iterations can not be determined at compile time. For example, the presence of flow-control commands such as break, return, or continue generally makes such analysis impossible. We have found no realistic examples where this restriction is a limitation, however.

4 Examples

The Lambertian example given so far is a very simple example of an *isotropic* lighting model, i.e. a surface that scatters light uniformly in every direction. Such lighting models can typically be efficiently implemented in model, world or view space. However, in most real-world surfaces there is an asymmetry, relative to the local surface orientation, in how light is reflected. In such *anisotropic* models, the local orientation is usually expressed by the tangent space of every point on the surface. Common examples of such models are *bump* [1] and *parallax mapping* [4], that uses tangent space to compute normal and, in the case of parallax mapping, texture space offsets to give appearance of surface wrinkles or bulges. Listing 3 gives an implementation of bump mapping in our language. Note the declaration of the new space `TangentSpace`, constructed from the basis vectors `T`, `N`, and `B`. Our compiler can use this space to infer that, in this case, the best space choice is tangent space.

For performance results of some common isotropic and anisotropic shading algorithms see Table 2. These algorithms demonstrate different levels of complexity, ranging from just a few scalar products (the Lambertian and Blinn-Phong) to complex, iterative algorithms that perform ray-tracing in texture space (Parallax mapping). All measurements compare a space-free implementation written in our prototype language to an equivalent, manually optimized, version implemented in GLSL. All hand-written examples contain implicit assumptions (e.g. that the view to model space transform is unitary) and explicit vector normalization in the pixel shader. All such assumptions are eliminated in the space-free variants. In all examples, the transforms and spaces inferred by our compiler precisely matches those used in the hand-written examples.

We see that the hand-optimized consistently out-perform the space-free shader variants. However, the performance gains of hand-optimizing a shader is only in the order of a few percent. This indicates that the space inference approach is useful in practice. The exact source of the performance disparity is difficult to pin-point since the GLSL compiler does not allow the generated assembler to be inspected. However, the most likely source is that hand-writing lets the programmer exploit instruction-level SIMD parallelism available on the GPU. Presumably, the structure of the code generated by our backend is less amenable to parallelization by the GLSL compiler.

5 Discussion

We have presented a novel domain-specific language for programming GPUs that enables shader programs to be written without explicit space transforms. This enables shading algorithms to be implemented in a more general and portable manner while allowing optimization to be performed on a per-application or even per-model basis. Consequently, space-free shaders can be reused in other applications without changes.

Shader	Performance (FPS) Space-free	Hand-written	Rel. perf.
Lambertian	1550	1580	98%
Blinn-Phong	1280	1310	97%
Bump mapping	1230	1240	99%
Refraction	917	950	96%
Parallax mapping	568	581	98%
View space bump mapping	-	1100	-

Table 2: Performance measurements of a simple synthetic scene running different shaders. The view space bump mapping is a hand written example demonstrating the performance impact of using an inappropriate space. Performance figures giving absolute performance rendered frames per second (FPS) of a synthetic scene. All measurements were run on an NVIDIA GeForce 8600 GT graphics card using a C++ rendering engine.

The performance of the generated GPU code is very close to that of hand-optimized versions which indicates that the approach given in this paper is useful in practice.

Much previous work has been devoted to translating shaders for off-line use, typically written in the RenderMan shading language [3], to real-time graphics hardware: reducing memory-accesses [7], automatic multi-pass shader factorization [8], and semi-automatic vertex/pixel factorization of shaders [9]. Our paper represent a previously unexplored avenue that could be used to further improve on these results.

Bibliography

[1] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 5, pages 286–292, 1978.

[2] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[3] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 289–298, 1990.

[4] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proceedings of the ICAT2001*, volume 12, pages 205–208, 2001.

[5] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[6] Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract shade trees. In *I3D '06*, pages 79–86, New York, NY, USA, 2006. ACM.

[7] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Graphics hardware*, pages 7–14. Eurographics Association, 2003.

[8] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 27, pages 425–432, 2000.

[9] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 159–170, 2001.

[10] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.

Paper VI

Separating shading from rendering in real-time graphics applications

Calle Lejdfors and Lennart Ohlsson
Department of Computer Science
Lund University
Lund, Sweden

{calle.lejdfors|lennart.ohlsson}@cs.lth.se

ABSTRACT

Programmable graphics processing units (GPUs) have become extremely powerful processors, capable of executing complex *shader programs* with high performance. To maximize the utilization of the GPU while maintaining a low CPU load, several shader-based rendering techniques are currently used. However, these techniques require shading computations (i.e. colorization computations) to be explicitly combined with code to handle the rendering technique itself. This results in code duplication and shader programs that are difficult to reuse and modify.

In this paper we present a framework for automatically transforming an existing shader to use any combination of the techniques instanced rendering, deferred rendering, or vertex stream-out caching. This reduces code duplication and increase code reuse. In experiments, performance of the transformed shaders is equal to that of hand-written code.

Manuscript in preparation

1 Introduction

Modern programmable *graphics processing units* (GPUs) provides an extremely high performing computational pipeline capable of producing near photo-realistic visual results at interactive frame rates. Currently, GPUs outperform CPUs by more than a magnitude and the relative performance increase is higher, meaning that this gap will continue to grow. GPUs are programmed using specialized *shader programs* that control the different pipeline stages to achieve a wide range of visual effects.

The key to high-performance rendering is efficient utilization of the GPU, typically with only minimal CPU overhead. In this paper we focus on some relatively recent rendering techniques for achieving efficient rendering: deferred shading [4], instanced rendering [17], and vertex stream-out caching. These techniques are shader-based and works by combining a shading algorithm, such as Blinn-Phong [1], with the chosen rendering technique into one or more shader program. By using programmable shader hardware in this way, it is possible to achieve very efficient rendering with minimal CPU intervention.

However, in all these techniques, the shading method must be explicitly combined with the rendering method in the shader programs. Hence, if an application to use both deferred and *direct* rendering, two or sometimes more different shader variants must be implemented. Each of these variants implement the same shading algorithm but using different rendering techniques. Furthermore, the rendering techniques can be composed, but constructing an instanced, deferred shader requires writing further variants. This becomes a problem both of managing and implementing these shaders.

In this paper we show that it is possible to automatically construct deferred, instanced and/or cached versions of an existing shader. This enables shaders to be reused irrespective of rendering method; a single shader can be used both in a direct and deferred context, using either instancing or not, without requiring any source code modifications. This reduces development times and results in shaders that are easier to maintain and modify (since rendering technique is not explicitly woven with the shading code).

The framework presented is fully configurable, allowing the user to decide whether data should be stored in a interleaved manner or not, the storage precision to use, etc. This enables render-method specific information to be kept separate from the shading algorithm, allowing an application to easily support multiple levels of precision suitable for different hardware and performance configurations. We have incorporated our framework into an existing rendering engine with results equal to that of using hand-written shaders.

The rest of the paper is organized as follows. In Section 2 we present the rendering techniques studied in this paper in more detail. In Section 3 we introduce our shader analysis and rewriting framework. Next, we demonstrate how our framework can be integrated into an existing renderer (Section 4), and some example shaders (Section 5).

Last, in Section 6 we summarize and discuss the contributions made in this paper.

1.1 Previous work

The first GPU generations where programmed using assembler languages closely tied to the underlying hardware. Currently however, GPUs are typically programmed using specialized *shader languages* such as Cg [8], HLSL [6], or GLSL [15]. These languages all provide C-like syntax with data types tailored specifically to the GPU. The choice of shader language depends on graphics API used: HLSL is tied to Microsoft's DirectX [2], and GLSL to OpenGL [16]. Cg may be used with both APIs.

Other systems for programming GPUs have been considered. Some consider translating Renderman or Renderman-like shaders to programmable hardware [14, 12]. Other take an embedded approach, enabling GPUs to be programmed in the same language as the rest of the application [9, 5]. However, no work known to the authors considers the problem addressed in this paper, i.e. that of automatically transforming shaders to support different shader-based rendering techniques.

2 GPU rendering algorithms

We will begin by giving a brief overview, including pros and cons, of the rendering techniques considered in this paper.

Instanced rendering Typically, when rendering a model the CPU submits the geometry information, along with the transformations matrices etc., every time a model is rendered. For example, suppose we want to render multiple spheres, all having the same tessellation level and radius, but each its own color and position. Using some shader this can be accomplished using a simple loop:

```
shader->bind();
Sphere* sphere = ...
for ( int i = 0 ; i < noSpheres ; i++ ) {
    shader->setUniform("model2world", sphere->modelTransform);
    shader->setUniform("color", sphere->color);
    sphere->render();
}
shader->release();
```

However, this is wasteful since there is a CPU overhead for each render call. Also, the sphere's geometry information must potentially be sent multiple times across the bus to the GPU.

Instanced rendering allows the geometry data to downloaded once and then used multiple times to render the other models. This technique relies on using a shader to distinguish between the different rendered mesh instances and choose the appropriate transformation matrix, for example, for that instance. The CPU needs only download

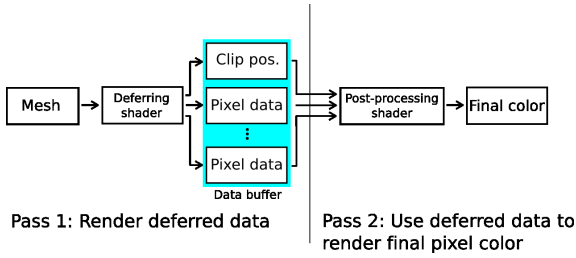


Figure 1: A pipeline overview of deferred shading

an array model-specific data such as transformation matrices, colors, etc., and then draw all the instances:

```
shader->bind();
shader->setUniform("model2worlds", modelTransforms);
shader->setUniform("colors", colors);
sphere->renderInstances(noSpheres)
shader->release();
```

The CPU overhead is lower than in the example above, and the mesh's geometry is only sent to the once the GPU. Instanced rendering can be combined with GPU-based skinning to draw large crowds of animated models very quickly, for example [10].

Deferred rendering In direct rendering each mesh is rendered using a particular shader to compute pixel color values that are written to the frame buffer. In deferred shading, instead of writing the resulting color value at each pixel, the information necessary to perform the shading is stored for each pixel in a *data buffer*[1]. Then, once all objects have been drawn, a shader processes each pixel of the deferred data buffer to determine the final pixel color. For an overview of deferred rendering see Figure 1. We will call the shader used in the first pass of deferred shading the *deferring shader*, and the shader in the second pass the *post-processing shader*.

The advantages of this approach is that only pixels that are visible are shaded, thus saving time when using expensive shaders, or when there is significant overdraw in the scene. Also, there is no need for CPU intervention to determine which light affects which meshes and, hence, lighting cost is independent of scene complexity [11]. The disadvantages are high bandwidth and memory requirements, and that only a single shader can be used for the whole scene. Also, hardware supported multi-sampling can not be used with deferred shading. Furthermore, transparent materials are difficult to incorporate into a deferred rendering framework. One possible technique is to use

[1]This is sometimes called a "fat" frame buffer since typically more than one 4-tuple of floating point data is stored per pixel. We will use the term data buffer in this paper, however.

depth peeling [7] but it is both expensive and complex. Hence, transparent objects are usually drawn using direct rendering after the deferred shading has been computed.

Vertex stream-out Previous GPU generations had a fixed pipeline where the output of each stage was fed into the next without being accessible to the programmer. Current GPUs allow the result of a vertex or geometry shader computation to be streamed to GPU memory before it reaches the pixel shader. This enables, for example, skinning computations to be cached to avoid expensive re-computations. This is particularly useful when the skinned model must be drawn multiple times, for instance, when affected by multiple lights or is casting multiple shadows. Another area where vertex stream-out is useful is when implementing particle systems and mesh deformations on the GPU.

3 The framework

The framework we present in this paper is capable of automatically transforming a shader to use any combination of instancing, deferred rendering, and stream-out caching. This enables shaders to be expressed irrespective of the rendering method used. This increases reusability and portability and significantly reduces code duplication, even in smaller projects.

Note that all the rendering methods studied here requires support in the application. The aim of this paper is to present a method to facilitate supporting different rendering methods at the shader level only. We will, however, discuss integration into a rendering engine in Section 4.

The framework consists of two parts: dependency analysis and program rewriting. The dependency analysis stage determines how a shader must be split to ensure that data dependencies are preserved. This is information is used as input to the rewriting stage that performs the actual shader transformation. The rewriting part of the framework can create new shaders, insert and remove instructions from existing shaders, as well as move operations between different pipeline stages.

The framework operates on an intermediate static single assignment representation [3] of the shader program. The framework was implemented in Python and uses a embedded shading language [9] that is similar to GLSL. Our language provides a slightly richer type system that distinguishes between different vector types (points, vectors, unit vectors, and directions). Having specialized types for directions and unit vectors enables us to automatically use the correct interpolation strategy for these quantities (normalize in the pixel shader only for directions, normalize in both vertex and pixel shader for unit vectors). Using language embedding avoids the additional work of implementing a full compiler front-end, thus enabling a quicker development and testing cycle. The back-end code generator creates GLSL code.

3.1 Instanced rendering

Consider the following vertex shader that computes the clip-space position of a vertex, using camera and model matrices:

```
uniform(Mat4, camera)
uniform(Mat4, model)

def main():
    gl_Position = camera*(model*gl_Vertex);
```

This shader can be used to draw objects using the more expensive loop-based method (see Section 2). Now consider the following shader that performs the same operations, but that can be used for instanced rendering:

```
uniform(Mat4, camera)
uniform(Array(Mat4, maxNoInstances), models)

def main():
    gl_Position = camera*(models[gl_InstanceID]*gl_Vertex);
```

Here, instead of using the model matrix directly, the shader is supplied with an array of matrices. It then uses the built-in index gl_InstanceID to access the appropriate matrix, depending on which model instance is currently being rendering.

Our framework can convert the first shader to the second. To do so user declares that the model matrix contains instance-specific data. Then, the rewriting stage will then change the declarations and all uses of these variables to arrays and array lookups, respectively. Hence, it will first replace the uniform declaration to an array instead. Then every use of the model matrix is replaced by a lookup of the instance-specific element in the models array. Also, the framework will automatically compute the maximal number of instances that the hardware can support (i.e. maxNoInstances) depending on the number of constant registers available.

3.2 Deferred shading

The instanced transformation above does not require complex dependency analysis. Transforming an existins shader into a deferred one is slightly more involved, however. Consider the Blinn-Phong shader in Listing 1, page 105. This shader computes the Blinn-Phong lighting contribution from at most 100 point lights shining on a surface. If we want to transform this shader to defer the lighting computation (as described in Section 2) we must deduce what information is necessary to shade a given pixel. That is, which variables are required by the post-processing shader to compute the final pixel color.

In the case of the Blinn-Phong shader, the variables that are needed are the viewer vector V, the surface color gl_Color, and the normal N. Together, they provide the necessary data to compute the lighting contribution.

```
uniform(Int, noLights)
uniform(Array(Struct(viewPos=Vec3, color=Vec3), 100), lights)
varying(Vector, N)
varying(Point, V)

def vertex():
    gl_Position = ftransform()
    N = normalize(gl_NormalMatrix*gl_Normal)
    V = direction(-gl_ModelViewMatrix*gl_Vertex)
    gl_FrontColor = gl_Color

def pixel():
    gl_FragColor = RGBA(0, 0, 0, gl_Color.a)
    for i in range(noLights):
        L = normalize(lights[i].viewPos + V)
        H = normalize(V+L)

        diffuse = saturate(dot(viewN, L))
        specular = pow(saturate(dot(viewN, H)), 8)
        surfaceC = diffuse*gl_Color + specular
        gl_FragColor.xyz += lights[i].color
```

Listing 1: A Blinn-Phong shader that supports multiple point light sources.

Our framework finds these variables in the following way; First, all operations that depend, directly or indirectly, on the light source information (i.e. the `lights` array) are located. Since the lighting computations are deferred, these operations must be performed in the post-processing shader. Next, the remainder of the operations (which can be computed without knowing anything about the light sources) are put in the deferring shader. Last, all terms that are defined in the deferring shader and are used at least once by the post-processing shader are found. These terms represent precisely the information that must be written to the data buffer.

After determining which operations should be put in which shader, and which terms should be stored in the data buffer, the framework constructs two new shaders that implement the deferring and post-processing. The name and storage type of the deferred terms can be queried by the application, allowing the appropriate frame buffer to be constructed for holding the deferred data. This enables the application to control the precision of the deferred shader (for more information see Section 4.1).

3.3 Vertex stream-out caching

Caching a vertex computation is a similar problem to the deferred shading above. The main differences is that information is stored per-vertex rather than per-pixel. Suppose we wish to cache the computation of the skinned position and normal of the matrix palette skinning example in Listing 2, page 110. Then, by reformulating, we

can describe this as the same problem as deferring computations depending on the gl_ModelViewProjectionMatrix, gl_ModelViewMatrix, and gl_NormalMatrix matrices.

Hence, we can use the above analysis algorithm to find those operations which should be performed in the *caching shader* (i.e. the shader responsible for computing the vertex data to be cached) and those that should be put in the *rendering shader* (i.e. the one using the cached data to render the model). Operations that depend on the above three matrices are scheduled to the rendering shader, the others are put in the caching shader. Similarly to the deferred shading above, all variables defined in the caching shader that are used in the rendering shader must be streamed out.

As above, after determining which operations should be put in which shader, our framework constructs the caching and rendering shaders. The data that should be cached can be queried by the application to allow the appropriate target buffers to be constructed.

3.4 Handling conditionals

When analyzing and rewriting shaders, special care must be taken when dealing with conditionals. For example, when transforming a shader to use deferred rendering, it is possible to have conditionals which must be split across the deferring and post-processing shaders. Consider the following code:

```
if gl_FrontFacing:
    N = normalize(gl_NormalMatrix*gl_Normal)
    L = normalize(light.viewPos + V)
    color = saturate(dot(N,L))*vec4(1,1,1,1)
else:
    color = vec4(1,0,0,0)
```

If computations depending on the light parameter are deferred, the color computation in the if-branch should be deferred also (since it depends on L which depends on light). However, the color in the else-branch does not depend on light and hence, should not be deferred.

We handle this case by mandating that every assignment to the same variable in a conditional must be marked (i.e. deferred or not deferred) in the same way[2]. Hence, in this example the generated code is

```
if gl_FrontFacing:
    vec3 N = normalize(gl_NormalMatrix*gl_Normal);
    ## write N to data buffer

## write gl_FrontFacing to data buffer
```

for the deferring shader. The post-processing shader will then look like:

[2]Or, equivalently, we mandate that both arguments to a SSA φ-function must be marked in the same way.

```
frontFacing = ## read from data buffer
if frontFacing:
    N = ## read from data buffer
    L = normalize(light.viewPos + V);
    color = saturate(dot(N,L))*vec4(1,1,1,1);
else:
    color = vec4(1,0,0,0);
```

Notice that we do not write N to the data buffer if the pixel is not front facing. This is not a problem however, since we also store the whether the pixel is front facing or not. If it is not, we will never read normal information back from the data buffer.

4 Application integration

As stated above, using deferred or instanced render methods will require some level of application support. It is not possible to use these techniques efficiently without providing the necessary structures in the rendering engine. For example, instanced rendering requires that all instances that should be drawn of a particular mesh be submitted at one time. This can be simplified using a suitable API, such as that proposed by Carucci (Chapter 3 of [13]). Similarly, deferred shading requires maintaining the deferred data buffer as well as implementing and integrating direct rendering in order to be able to use particle systems and other, transparency-based, effects.

4.1 Performance and precision tuning

When using instanced rendering the performance is roughly proportional to the number of instances drawn in a single call. Hence, the main performance benefit comes using as many instances per draw call as possible, to avoid pipeline stalls. When using deferred shading however, performance and precision becomes important factors to consider. For instance, the precision of the data buffer will greatly affect both the required storage and bandwidth requirements, as well as the visual quality of the result.

Our framework gives complete control to the programmer, enabling the precision and storage method of each item written to the data buffer to be specified. For example, consider the shader in Listing 1. If we wish to store the normal ith frame buffer layer using a scaling transform to map it into the unit cube $[0..1, 0..1, 0..1]$ we do

```
shader.overrideDeferred("N", i, "0.5*N+0.5", "2*N+1");
```

This will transform the normal to the unit box when writing it to the frame buffer, and then back to the unit sphere when reading it from the frame buffer. This is useful when using lower-precision rendering targets that implicitly clamp their values, for instance.

The ability to override how individual elements are deferred is also useful when using multiple shaders in a single scene. Then the deferred output of every shader

Shader	Render method	Performance (FPS) w/ method	w/ direct
Blinn-Phong (1 light)	Instancing	5.9	3.1
Blinn-Phong (100 lights)	Deferred	32.6	9.5
Skinning + B-P (1 lights)	Caching	43.5	34.4
Skinning + B-P (100 lights)	Caching + Deferred	9.0	3.1

Table 1: Performance figures for some example shaders. The direct rendering approach uses GPU vertex buffer objects for geometry data which are submitted once for every object. The instancing examples uses the instanced rendering interface provided by GL_EXT_draw_instanced. Caching shaders uses OpenGL 2.1 buffers, and deferred shaders uses 128 bit floating-point storage for the intermediary data buffers. All measurements are made on a OpenGL graphics engine running on a GeForce 8600 GTS card with 512 MB of video memory.

must be matched to allow the post-processing stage to be executed. In this case, explicit specification of what output should be written to which layer is required.

5 Examples

We have tested our framework by transforming several different shaders to use deferred or instanced rendering, as well as vertex stream-out caching. The performance results are displayed in Table 1. It shows the measured performance in rendered frames per second when using the shader constructed by our rewriting framework to use the indicated rendering method. The result of using direct rendering using the original shader is also shown. In all examples, hand-written shaders using the same rendering technique and shading code were implemented in GLSL. The performance of our generated shaders was, in all cases, identical to their hand-written counterparts.

In the first example we render a test scene consisting a 10000 small objects (11000 triangles each) using Blinn-Phong shading with a single light source. Here, using instancing results in a speed up of about 2 times. The second example consists of 100 higher tessellated objects (48000 triangles) rendered using 100 point light sources. Using deferred shading results in a large performance gain ($\times 3.4$ times).

Next two examples uses a GPU skinning implementation with a single light source, and 100 light sources, respectively. Using caching here results in a speed-up of about 25% in the first example. By combining caching and deferred shading in the second example, the performance gain is roughly 3 times.

When combining caching with instancing we did not notice any performance increase compared to only using caching. Presumably, the fact that mesh data already resided in a GPU-side vertex buffer limits the usefulness of instancing in this case

(where the performance is obtained primarily from avoiding transferring data from the CPU to the GPU multiple times).

6 Discussion

We have presented a framework for shader data dependency analysis and rewriting that enables shading and rendering to be separated. We have shown that this framework enables rendering methods such as deferred shading or instanced rendering to be used without rewriting the original shader. This reduces code duplication significantly. Also, reusability and portability is increased since shading is not tangled with code specific to the render technique used.

```
attribute(IVec2, weightIndex)
varying(Vec4, viewPos)
varying(Vec3, viewN)
uniform(Float, frame)
uniform(Sampler2DRect, weights)
uniform(Sampler2DRect, positions)
uniform(Sampler2DRect, orientations)

## quaternion operations
def qmul(q1=Vec4, q2=Vec4): ...
def qconj(q=Vec4): ...
def qrot(q=Vec4, p=Vec3): ...

def vertex():
    position = vec3(0,0,0)
    normal = vec3(0,0,0)

    start = weightIndex.x
    end = start + weightIndex.y
    for i in range(start, end):
        tmp = texture2DRect(weights, vec2(i, 0))
        wp = tmp.xyz
        joint = tmp.w

        tmp = texture2DRect(weights, vec2(i, 1))
        wn = tmp.xyz
        value = tmp.w

        jp = texture2DRect(positions, vec2(joint, frame)).xyz
        jq = texture2DRect(orientations, vec2(joint, frame))

        position += (jp + qrot(jq, wp))*value
        normal += qrot(jq, wn)*value

    p = point(position)
    gl_Position = gl_ModelViewProjectionMatrix*p
    viewPos = gl_ModelViewMatrix*p
    viewN = normalize(gl_NormalMatrix*normal)
    gl_TexCoord[0] = gl_MultiTexCoord0
```

Listing 2: A skinning vertex shader program. This program can be used together with the view-space Blinn-Phong pixel shader in Listing 1, for example.

Bibliography

[1] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 5, pages 286–292, 1978.

[2] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, October 1991.

[4] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22(4):21–30, 1988.

[5] Conal Elliott. Programming graphics processors functionally. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 45–56, 2004.

[6] Kris Gray. *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[7] Baoquan Liu, Li-Yi Wei, and Ying-Qing Xu. Multi-layer depth peeling via fragment sort. Technical Report MSR-TR-2006-81, Microsoft Research Asia, 2006.

[8] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[9] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[10] Erik Millan and Isaac Rudomin. Impostors and pseudo-instancing for gpu crowd rendering. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 49–55, New York, NY, USA, 2006. ACM.

[11] Huvert Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2007.

[12] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 27, pages 425–432, 2000.

[13] Matt Pharr and Randima Fernando, editors. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[14] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 28, pages 159–170, 2001.

[15] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.

[16] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide*. Addison-Wesley Proffesional, 5th edition, 2005.

[17] Jeremy Zelsnack. GLSL psuedo-instancing. Technical report, NVIDIA Corporation, 2004.