Techniques for implementing embedded domain specific languages in dynamic languages

av

Calle Lejdfors

Akademisk avhandling

som för avläggande av teknisk licentiatexamen vid tekniska fakulteten vid Lunds Universitet kommer att försvaras offentligt i hörsal E:C, sektionen för elektroteknik och datateknik, Lunds Tekniska Högskola, onsdagen den 8:e mars 2006, kl. 10:15.

Organization LUND UNIVERSITY Dept. of Computer Science Box 118 SE-221 00 Lund Sweden	Document name LICENTIATE DISSERTATION Date of issue March 8, 2006 Sponsoring organization
Author(s) Calle Lejdfors	

Title and subtitle

Techniques for implementing embedded domain specific languages in dynamic languages

Abstract

Computer programming should be expressing the complicated in easily understandable parts. General languages provide tools and abstractions that allow many different problems to be formulated and solved. Unfortunately, these abstraction only rarely match the problem area precisely, resulting in solutions that are obscured by the need for support code unrelated to the problem.

This can be avoided, by constructing a language tailored to the specific problem, solutions can be expressed more clearly, resulting in programs that are easy to read, debug, and maintain. Designing a new language from scratch, however, is a costly venture, which has hindered the adoption of such domain specific languages (DSLs).

In this thesis, we explore and describe methods for constructing and implementing DSLs. By using an existing host language, DSLs can be implemented as embedded languages, inheriting the syntax and capabilities of the host language. We argue that the embedded language approach to implementing DSLs gives many benefits over traditional language implementation techniques. Furthermore, by using a dynamic programming language as host, we can take advantage of the high-level nature of such a language; both to facilitate the implementation of the DSL and to provide high-level programming constructs in the DSL.

This thesis describes two prototype languages; PyFX and PyGPU, both embedded in Python, for programming and controlling different aspects of modern graphics cards. PyFX is a language for constructing real-time graphics effects. It views effects as active entities, allowing them to be expressed concisely and clearly, without introducing the glue-code requirements of previous approaches.

The second language, PyGPU, is targeted at writing high-level image processing algorithms that execute on the high performing graphics processing unit (GPU) present on modern graphics cards. PyGPU uses a combination of a high-level image abstraction and a novel compilation strategy to translate high-level PyGPU code to efficient native code running on the GPU.

Key words

Domain specific languages, embedded languages, dynamic languages, computer graphics, graphics processing units

Classification system and/or index terms (if any)		
Supplementary bibliographical information		Language English
ISSN and key title 1652-4691		ISBN
Recipient´s notes	Number of pages 96	Price
	Security classification	

Distribution by (name and address) Department of Computer Science, Lund University
Box 118, SE-221 00 Lund, Sweden

I, the undersigned, being the copyright owner of the abstract of the above-mentioned dissertation, hereby grant to all reference sources permission to publish and disseminate the abstract of the above-mentioned dissertation.

Signature [illegible] Date February 14, 2005

Techniques for implementing embedded domain specific languages in dynamic languages

Calle Lejdfors
Department of Computer Science
Lund University

LUND INSTITUTE OF TECHNOLOGY
Lund University

ISSN 1652-4691
Licentiate Thesis 5, 2006
LU-CS-LIC:2006-2

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

Abstract

Computer programming should be expressing the complicated in easily understandable parts. General languages provide tools and abstractions that allow many different problems to be formulated and solved. Unfortunately, these abstraction only rarely match the problem area precisely, resulting in solutions that are obscured by the need for support code unrelated to the problem.

This can be avoided, by constructing a language tailored to the specific problem, solutions can be expressed more clearly, resulting in programs that are easy to read, debug, and maintain. Designing a new language from scratch, however, is a costly venture, which has hindered the adoption of such *domain specific languages* (DSLs).

In this thesis, we explore and describe methods for constructing and implementing DSLs. By using an existing *host* language, DSLs can be implemented as *embedded* languages, inheriting the syntax and capabilities of the host language. We argue that the embedded language approach to implementing DSLs gives many benefits over traditional language implementation techniques. Furthermore, by using a *dynamic* programming language as host, we can take advantage of the high-level nature of such a language; both to facilitate the implementation of the DSL and to provide high-level programming constructs in the DSL.

This thesis describes two prototype languages; PyFX and PyGPU, both embedded in Python, for programming and controlling different aspects of modern graphics cards. PyFX is a language for constructing real-time graphics effects. It views effects as active entities, allowing them to be expressed concisely and clearly, without introducing the glue-code requirements of previous approaches.

The second language, PyGPU, is targeted at writing high-level image processing algorithms that execute on the high performing graphics processing unit (GPU) present on modern graphics cards. PyGPU uses a combination of a high-level image abstraction and a novel compilation strategy to translate high-level PyGPU code to efficient native code running on the GPU.

Preface

This thesis is for the Licentiate degree, a Swedish degree between the MSc and PhD. It consists of an introductory part and four papers.

The research papers included in this thesis are:

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

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

iii. Calle Lejdfors and Lennart Ohlsson. Implementing an embedded GPU language by combining translation and generation. *To appear in SAC'06 Programming Language track*, 2006.

iv. Calle Lejdfors and Lennart Ohlsson. PyGPU: A high-level language for high-speed image processing. *Submitted for publication*, 2006.

Acknowledgements

The work presented in this thesis has been carried out within the *Computer graphics* group at the Department of Computer Science, Lund University. I wish to thank my supervisors Professor Boris Magnusson, Assistant Professor Lennart Ohlsson, and Assistant Professor Görel Hedin. In particular, I wish to thank Lennart Ohlsson for introducing me to Python and functional programming, as well as many discussions on the design and construction of software in general.

Furthermore, I wish to thank Jon Hasselgren for many rewarding discussions on graphics programming, game engine architecture, plants and other things. Mathias Haage, thank you for your valuable suggestions and encouragement. Also, the rest of the computer graphics group (Tomas, Jacob, and Petrik) have my heartfelt thanks, together with the rest of the department.

Finally, I wish to thank my family for always encouraging me to seek my own path in life (and for putting up with me when I do). Naturally, to all my friends, thank you for reminding me about what is important in life.

Contents

Introduction

The solution to a complex problem is usually to express it as a combination of smaller parts, where each part is of lower complexity. General programming languages provide facilities such as classes and objects, functions, or even macros, for describing and encapsulating these parts. However, expressing a given problem of some domain in a general language involves translating the problem into the syntax and capabilities of that language. This involves writing "support code" unrelated to the problem at hand. Nevertheless, this code is required to express the problem. As a result, the structure of the solution can be obscured by the need to describe it in terms understandable to the programming language used.

This problem can be addressed by constructing a *domain specific language* (DSL) targeted specifically at solving problems within a given domain. By construction, a DSL can be made to fit the demands of the problem domain precisely, creating an "ultimate abstraction" [29] for solving problems in the domain. By using a dedicated language, programs can be expressed using the language and idioms of the problem domain instead. The DSL compiler can perform validation and optimization at the domain level, resulting in fast and correct programs.

Designing and implementing a new language from scratch is a difficult and costly task. Luckily, we already have access to a number of well-designed general languages. By using a general language and adapting it to the problem domain, we can construct an *embedded* domain specific language. By reusing parts of the syntax and semantics of the *host* language, the DSL-implementor is free to focus on the semantic details of the problem domain rather than low-level tasks such as code generation and type checking. This reduces the complexity and time-requirements of implementing a DSL.

This thesis deals with the construction and implementation of embedded DSLs in, so called, *dynamic languages*. Dynamic languages, such as Python, Ruby, LISP, and Smalltalk, provide a rich set of high-level programming constructs, combined with advanced features such as introspection, the ability of a program to observe its own execution, and dynamic code execution. This makes dynamic languages well-suited for implementing embedded DSLs, enabling high-level programming for both DSL users and implementors. In this thesis we show how introspection and dynamic code execution can be used to enable straightforward construction of high-level domain specific embedded languages.

We have implemented two prototypes languages embedded in Python, both targeted at programming different aspects of programmable graphics cards. The first one is PyFX, a language for constructing real-time graphics effects. It uses introspection to enable viewing effects as *active* entities. Doing so reduces the amount of glue-code needed, compared to previous effect frameworks, such as FX and CgFX. Also since PyFX is a complete programming language, effects can be described more concisely and clearly than previously possible.

The second prototype language is PyGPU, a DSL for writing high-level image processing algorithms that execute on the high performance graphics processing unit (GPU) of modern graphics cards. PyGPU uses a combination of generative and translative techniques to compile high-level PyGPU code to the GPU. This combined strategy, together with a high-level functional image abstraction, enables PyGPU to make a large subset of Python available to the DSL programmer. The compiler of PyGPU uses introspection and dynamic code execution, allowing it to be concisely implemented by reusing large parts of the existing functionality in the Python interpreter. The GPU code generated performs very well, showing that it is possible to combine high-level programmability with high performance.

The rest of this introduction is organized as follows: Sections 1, 2, and 3 give background information on domain specific languages, dynamic languages and graphics programming, respectively. Section 4 summarizes the contributions made in this thesis and Section 5 concludes this introduction and discusses some ideas for future work.

1 Domain specific languages

The earliest uses of domain specific languages are compiler-related tools such as parser and scanner generators (Lex [38], YACC [33], and Zephyr [58]) and in compiler construction (JastAdd [26], and UUAG [2]). Other examples of DSLs are database query languages (SQL [9], dBase [34]), symbolic algebra (Maple [10]), and parallel programming (Orca [4]). Also, languages such as LISP [51] and FORTRAN [3] 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. [55] propose 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 [40] 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. Some domain specific language are also called "little languages" [5].

Using a language specialized for the problem at hand comes with a number of benefits. DSLs allow problems to be expressed using the idioms inherit 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 that which 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 [28], 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) [29], 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. The first example of an embedded DSL is the simulation support of SIMULA [15].

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 (CLOS) [37], for object-oriented programming in LISP, and Verischemelog [32], 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 [35], 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 [19] and Yampa [30] (reactive animations), Haskore [31] (music), and Lava [6] (hardware description and verification). The industry standard C++ [52] has also been used to implement embedded languages; for example TBAG [20], for interactive animations, and Spirit [50], a template meta-programming language [14] for constructing parsers.

When the target is *not* the same as that of the host language then the above approaches cannot be used. Instead a *generative* approach is typically used, 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 [18], image manipulation and synthesis, and Vertigo [17], 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, respectively. In C++, Sh by McCool et al. [41] is a language for programming GPUs. It uses a method where functions are executed and their execution is recorded. This recording is then used as input to a compiler responsible for generating GPU assembler code.

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 be in-lined in the recorded stream of operations. Consequently, a host language conditional will be translated to conditional compilation in the embedded language.

2 Dynamic languages

Dynamic languages, such as Python [45], Ruby [54], Dylan [13], LISP, Scheme, and Smalltalk [22], are languages designed to increase programmer efficiency. Dynamic languages enables 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 lead to an alternate name for dynamic typing: *duck typ-*

ing, inspired by the metaphor "if it looks like a duck and quacks like a duck, it must be a duck" [56]. These features together with the fact that dynamic languages tend to be higher-level than traditional static ones, 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 is not known statically, generating efficient native CPU code for dynamic languages is a very difficult problem [47, 48].

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, C# [27], and the C++-extension OpenC++ [12], also supports 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 browsers and 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 [57], and aspect-oriented programming features [53]. Also, 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 3D 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) which enables parts of the rasterization process to be programmatically redefined. For a thorough introduction to programming graphics cards see "The Cg Tutorial" by Fernando and Kilgard [46] or the "OpenGL Programming Guide" by Shreiner et al. [49].

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

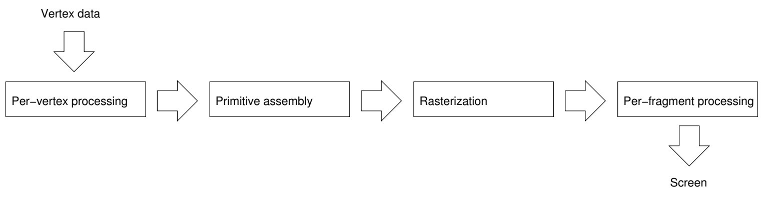


Figure 1: The graphics pipeline

either on or off. Also, the different parts can be controlled to some limited degree. For instance, the depth testing part of the fragment processing stage can be turned on or off, and be instructed to use different functions for determining whether a fragment 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, anti-aliasing, 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 [49] and DirectX [16], are supported by all major graphics card vendors.

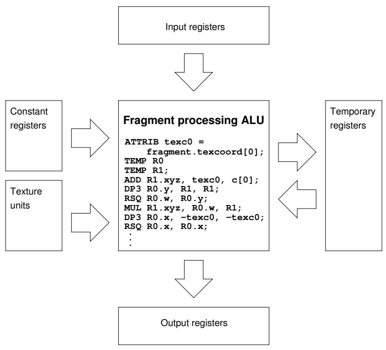
3.2 The graphics processing unit

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 fixed-function pipeline. Examples of such extensions are per-fragment lighting, bump-mapping, and vertex weighting. As an interesting comparison, today there are over 300 different OpenGL extensions. A more general method for handling extensions was needed.

This led to the introduction of programmable graphics hardware, commonly called the *graphics processing unit* (GPU). The GPU allows the vertex and fragment processing steps 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.

Programs running on the GPU are *shader programs*, referring to the act of coloring, or *shading*, a pixel. GPU programs that operate on vertices are called *vertex shaders* and programs operating on fragments, consequently, *fragment shaders*.[1]

[1]In DirectX this is known as *pixel shaders* because it is the part of the shader that is responsible for computing the final pixel color. The OpenGL community use the term fragment instead, since it is a complete data fragment (including, for instance, color, texture coordinates, and depth information) that is computed in this part of the pipeline. If this fragment passes all the per-fragment tests it is written as a color value to a pixel. The distinction is unimportant in the context of this work and the term fragment shader will be used from here on.



```
ATTRIB texc0 =
    fragment.texcoord[0];
TEMP R0
TEMP R1;
ADD R1.xyz, texc0, c[0];
DP3 R0.y, R1, R1;
RSQ R0.w, R0.y;
MUL R1.xyz, R0.w, R1;
DP3 R0.x, -texc0, -texc0;
RSQ R0.x, R0.x;
 .
 .
 .
```

Figure 2: Schematic overview of the fragment processing unit of the GPU

The GPU programming model

Modern GPUs are highly parallel stream processors capable of raw floating-point performance in excess to that of current generations CPUs. A typical GPU uses a SIMD-type architecture, where operations are performed in parallel on 4-tuples of floating-point values. For an overview of the fragment processing stage 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. Current generations do not provide globally writable memory. Memory may be accessed only by reading from 1, 2, or 3-dimensional texture maps. Instead computations are performed using a set of registers.

This 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 fragment program may only write information to one pixel on the screen. The position of this pixel is determined earlier in a fixed-function part of the pipeline and cannot be changed. This lack of *scattered* writes has implications on using the GPU for more general, non-graphical computational tasks.

GPU programming languages

The first generations of GPUs were programmed using assembler languages specific to each vendor. Now, NVIDIA's Cg [39], Microsoft's HLSL [24], and GLSL by the OpenGL ARB [36], provide C-like languages for programming the GPU. These enable shader programs to be written at a higher level than previously possible. Of these languages, HLSL and GLSL are specific to DirectX and OpenGL, respectively. Cg can be used with both DirectX and OpenGL.

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

3.3 Effects

The common use of shaders is for implementing visual effects, such as realistic lighting models, shadows, glowing objects, furry objects, etc. In these scenarios, the shader programs usually depend both on non-programmable pipeline states being set, as well as on CPU-level code for setting program parameters and downloading texture information. Furthermore, many visual effects require rendering the same geometry multiple times, in so called *passes*, using different shaders or shader parameters in each pass.

This lead to an extension of the above GPU languages called *effects*, or *fx*-files. Examples of effect frameworks are FX [24] for HLSL, and CgFX [8] for Cg. Effects enable shader programs to be associated with their corresponding pipeline states into passes. Multiple passes can then be combined to create *techniques* that encapsulates a complete visual effect. An fx-file may have multiple techniques each containing alternate effect implementations suitable for different hardware capabilities, different levels of visual acuity, etc. Also, effects have parameters, known as *tweakables*, that allow the effect to be controlled. Examples of tweakables are material reflection properties, fur length, and glow amount and intensity.

Effect files solve the problem of associating shaders with pipeline states. However, they only partially address the problem of encapsulating effects: CPU-based code necessary for the effect, such as the computation of a model-relative light position, cannot be expressed within the effect framework.

3.4 General purpose GPU

The programmability and high performance of the GPU have spawned a great interest in using it for computationally intensive, non-graphical tasks. Algorithms, such as image processing [43], fluid simulation [25], behavioral models [11], audio processing [59], database operations [23], and numerical solvers [21], have all been implemented to run on the GPU. Using the GPU for a non-graphical task, such as simulation or numerical computations, is collectively known as *general purpose* GPU, 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.

However, as explained in Section 3.2, the programming model of the GPU is quite restricted. Furthermore, since the GPU only works with graphic primitives such as vertices, triangles, and textures, porting an algorithm to run on the GPU entails expressing it in terms of these primitives. 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, real-time graphics programming. These limitations have severely limited the spread and uptake of GPGPU techniques.

Some projects have been proposed for facilitating the use of the GPU for numerical processing. BrookGPU [7] 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 [42] provide similar template-based abstractions for implementing GPGPU algorithms.

4 Contributions

This thesis considers the area of implementing embedded DSLs in dynamic languages. The key contributions are:

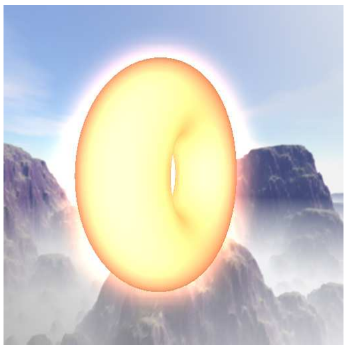
- A method for compiling high-level, embedded DSLs suitable for dynamic languages.
- An image abstraction suitable for translation to the GPU.
- An introspective method for reducing glue-code requirements when using domain specific languages.

This work has been carried out in the context of programming modern graphics cards, including both controlling pipeline states and high-level programming of the graphics processing unit. The results in this thesis were obtained by designing and implementing two novel prototype languages; PyFX and PyGPU, both embedded in Python.

4.1 PyFX

PyFX is a language for writing multi-pass, real-time visual effects. It makes extensive use of introspection in order to minimize dependencies between effects and the application that use them. Previous effect frameworks break encapsulation by introducing cyclic dependencies between application and effect. PyFX avoids this by viewing effects as active objects that can automatically extract information from the application. This substantially reduces the amount of glue-code required, often down to just a couple of lines. Also, since PyFX is a real programming language, rather than a simple file format, it enables common software design methodologies, such as modularization and abstraction, to be used in the construction of effects.

PyFX supports run time composition of shader code, enabling efficient code sharing and reuse, as well as allowing more flexible uses of effects. For instance, using shader composition in PyFX allows effects to be used with any combination of light sources and material properties. In addition, PyFX introduce an additional set of effect primitives, such as render-to-texture and image processing support, not found in other effect frameworks at the time. An example effect together with its implementation is shown in Figure 3.



```
GlowColor = (0.9,0.5,0.1)

blurBuffer = RGBABuffer()

def Gaussian2D(buffer):
    return 3*[Gaussian1D(buffer,(1,0)),
              Gaussian1D(buffer,(0,1))]

technique = [
    Render(),
    Render(Target=blurBuffer,
           Color=GlowColor),
    Gaussian2D(blurBuffer),
    AdditiveBlend(blurBuffer)]
```

Figure 3: A PyFX glow effect example. The effect renders the geometry to an off-screen buffer. This buffer is blurred using consecutive 1-dimensional convolutions and the result is blended onto original geometry, resulting in a glowing, halo-like appearance.

The implementation of PyFX is described in detail in Paper 1. Paper 2 provide a number of usage examples of PyFX, focusing on the benefits of using a real programming language for describing effects.

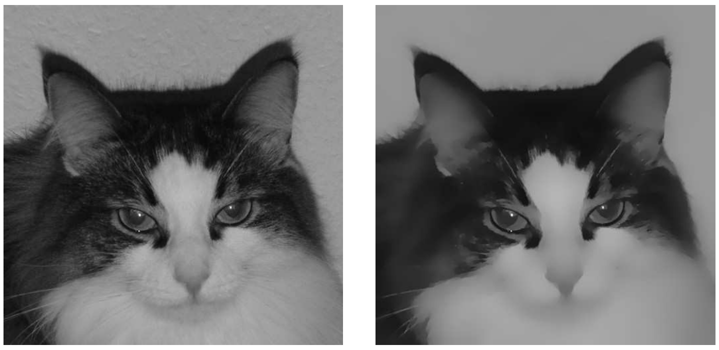
4.2 PyGPU

PyGPU is a high-level embedded language for writing image processing algorithms that run on the GPU. The GPU is a very restricted platform whose lack of writable memory makes direct translation of, for example, lists impossible.

The PyGPU language is designed around an abstraction viewing images as functions from a 2-dimensional grid to some space of colors. This abstraction encodes the capabilities and restrictions of the GPU at the domain level, efficiently disallowing those algorithms which cannot be translated to the GPU. The abstraction is still powerful enough to allow many interesting image processing algorithms to be implemented to take advantage of the GPU. In fact, PyGPU enables most of Pythons features to be used in the construction of image processing algorithms. See Figure 4 for an example of Perona-Malik [44] anisotropic diffusion implemented in PyGPU.

The PyGPU compiler uses a combination of translative and generative techniques to translate algorithms to efficient GPU code. This enables many of the high-level features of Python to be used in the construction of GPU programs. PyGPU makes heavy use of the introspection and dynamic code execution facilities of Python, allowing the compiler to be expressed concisely making maximal reuse of the existing Python interpreter and run time system.

The implementation of the PyGPU compiler is described in Paper 3. A number of algorithm examples and performance characteristics are discussed in Paper 4.



```
@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 4: Perona-Malik anisotropic diffusion in PyGPU. The diffusion step is written concisely as the sum of the contributions from the nearby pixels. The function g and the variable K controls the slope of the diffusion transport coefficients. The image above was generated with $K = 0.25$ using 400 iterations.

5 Conclusions

Dynamic languages have shown to provide a flexible environment for experimenting with and implementing embedded languages. Support for introspection and dynamic code execution have been shown to greatly facilitating implementing both ordinary, library-based embedded languages as well as those that are translated to some platform or processor.

Using introspection to compile parts of a program for some other processor, as is done in PyGPU, should be reusable in a variety of scenarios. For example, in the generation of efficient numeric CPU code and construction of program for embedded devices. Another interesting research venue would be extending PyGPU to handle more general GPGPU algorithms.

Bibliography

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

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

[3] 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:adr, 1957. pub-IRE. The online edition of the Oxford English Dictionary cites this as the second earliest mention of the name FORTRAN, with the extract "The programmer attended a one-day course on FORTRAN and ... then programmed the job in four hours using 47 FORTRAN statements.".

[4] 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.

[5] Jon Bentley. Little languages. In *Communications of the ACM*, volume 29(8), pages 711–721, August 1986.

[6] 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, New York, NY, USA, 1998. ACM Press.

[7] 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.

[8] CgFX 1.2 Overview. `http://developer.nvidia.com/`.

[9] 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, New York, NY, USA, 1974. ACM Press.

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

[11] 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.

[12] 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, New York, NY, USA, 1995. ACM Press.

[13] Iain D. Craig. *Programming in Dylan*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

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

[15] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[16] DirectX SDK Documentation. `http://msdn.microsoft.com/`.

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

[18] 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, London, UK, 2000. Springer-Verlag.

[19] 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.

[20] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. Tbag: a high level framework for interactive, animated 3d graphics applications. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 421–434, New York, NY, USA, 1994. ACM Press.

[21] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *To appear in Proceedings of the 2005 ACM/IEEE Super Computing Conference. November 12-18,*, 2005.

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

[23] 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, New York, NY, USA, 2004. ACM Press.

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

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

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

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

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

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

[30] 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.

[31] 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.

[32] 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, New York, NY, USA, 1999. ACM Press.

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

[34] Edward Jones. *The dBase language reference*. Osborne/McGraw-Hill, Berkeley, CA, USA, 1990.

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

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

[37] 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, Cambridge, MA, USA, 1993.

[38] 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, New York, NY, USA, 1979.

[39] 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.

[40] Matlab. `http://www.mathworks.com/`.

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

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

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

[44] 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.

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

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

[47] 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, New York, NY, USA, 2004. ACM Press.

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

[49] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide*. Addison-Wesley Proffesional, 4th edition, 2003.

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

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

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

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

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

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

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

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

[58] 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, Berkeley, CA, October 15–17 1997. USENIX Association.

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

Paper I

PyFX – An active effect framework

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

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

Abstract

The programmability of modern graphics processing units (GPUs) provide great flexibility for creating a wide range of advanced effects for interactive graphics. Developing such effects requires writing not only shader code to be executed by the GPU but also supporting code in the application where the effect is to be used. This support code creates dependencies between effects and the applications that use them, making it harder to evolve applications and to reuse effects. Existing effect frameworks, such as DirectX Effects and CgFX, can only provide partial encapsulation because they consider effects as passive data structures. In this paper we present an effect framework written in an ordinary scripting language where effects are active entities. This makes it possible to completely encapsulate both shaders and support code thereby minimizing the dependencies to the application.

SIGRAD 2004 Conference proceedings, pages 17–24, 2004

1 Introduction

The availability of programmable graphics processors has made procedural effects a key ingredient in real-time graphics productions. Where content creation previously was mainly the combination of a wide range of different kinds of artwork such as geometric models, textures, and motion data, it now also has to include algorithmic development. Writing the shader code to be executed on the graphics processors is something which traditionally is not part of an artist's skill set. Instead this new development model requires a closer relationship between artists and shader programmers. Previously programmers of interactive graphics applications were primarily concentrated with loading and displaying content created by the artists in an efficient and correct manner, a task which is handled fairly independent of the actual content. But with programmable graphics processors the roles of artists and programmers become more intertwined. When the artist conceives of a visual effect it is the programmers job to supply shader programs and the necessary modifications to the application for achieving that effect. But once written, the shader program typically requires actual textures and parameter values and it is the artists job to supply that.

For efficient collaboration it is important, to both artists and programmers, that the graphical effect is a well-defined entity. It should include all relevant resources and functionality, both shader code and application support, required for correct operation. This need for encapsulation is the motivation behind technologies such as the DirectX Effects by Microsoft and CgFX by NVIDIA. In these frameworks the notion of an *effect* is used as the key unit of abstraction. But although these technologies provide a number of features which improve the handling of effects they still require a substantial amount of application support. All but the most trivial effects have dependencies in the application that use them.

The effect framework presented in this paper aims to provide *complete encapsulation* of effects in the sense that specific support code avoided and parameter passing is made with the most unobtrusive mechanism possible. We have implemented a prototype which uses Python both for the implementation of the framework and to express the effects themselves. This paper is focused on the implementation of the framework and its application interface, whereas the benefits of writing effects in Python is described in more detail elsewhere [11].

1.1 Related work

The focus on complete effects is different from most other approaches. Most real-time shading language research has been focused on mapping high-level shading languages to real-time shading hardware. Research was initiated by Cook [3] with the introduction of shade trees, which spawned a number of shading languages such as RenderMan [8] or Perlins image synthesizer [18]. These languages were originally used for off-line shading but with advances in hardware Peercy et al showed that it was possible to execute RenderMan shaders on an extended OpenGL 1.2 platform by viewing the graphics hardware as a SIMD pixel processor [17]. Olano et al presented an alternative approach with the pfman language [16] for the Pixelflow rendering system [15], a flexible platform based on image composition which, unfortunately, bears little resemblance to the GPUs of today.

The computational model of separating per-vertex and per-pixel computations was introduced by Proudfoot et al [19] which allowed the to efficiently map shader programs to hardware. This separation is used explicitly the in real-time shading languages used in the industry today: Cg by NVIDIA [12], HLSL by Microsoft [7] and OpenGL shading language [10] introduced with OpenGL 2.0. This is also the case with the Sh language [13, 14]; a shading language embedded in C++ which provides a number of powerful high-level features for shader construction. Another embedded shading language is Vertigo [5] which uses the purely functional language Haskell as a host language to provide a clean model for writing shaders for generative geometry.

All these efforts have focused on various aspects of shader programming but the writing of effects containing multiple shaders have not received the same amount of attention. The Quake3 shader model [1] provides a rudimentary interface for controlling the application of multiple textures. DirectX Effects [4] extend the HLSL shading language and introduce a richer, more powerful interface for controlling the rendering pipeline. NVIDIA provide a superset of this functionality with their CgFX framework [2], based on Cg. This is further elaborated on in Section 1.3.

1.2 Shader programming

To demonstrate the issues involved in the implementation of shader based effects and how an active framework like PyFX can alleviate these problems we will use a running example throughout this paper. The description of this example will be fairly detailed because the causes of application dependencies and need for support can often be found in those details which would usually be omitted in a more concise description. The example we use is the lighting model known as hemispheric lighting, where the idea is to give a contribution of indirect light as a mixture of light from the sky and light from the ground. A given point is colored depending on orientation of its surface normal, the more it is oriented towards the sky the more light from the above light source it receives, and vice versa. The effect of using this model can be seen on the bunny in Figure 1.1. A shader program which implements this model can be written in Cg as

```
void main(float4 position : POSITION,
          float4 normal   : NORMAL,

      out float4 clipPosition : POSITION,
      out float3 color        : COLOR,

  uniform float4x4 ModelViewProj,
  uniform float4x4 ModelViewIT,
  uniform float4x4 WorldView,
  uniform float3   MaterialColor,
  uniform float3   SkyColor,
  uniform float3   GroundColor)
```

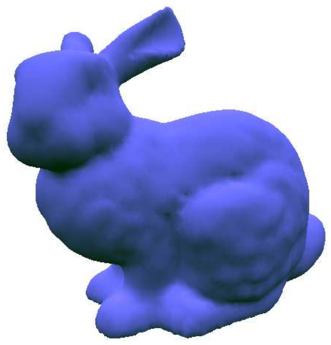


Figure 1.1: Hemispheric lighting on bunny

```
{
  clipPosition = mul(ModelViewProj, position);
  float4x4 ModelWorldIT = mul(WorldView,ModelViewIT);
  float3 worldNormal = mul(ModelWorldIT,normal).xyz;

  worldNormal = normalize(worldNormal);
  color = lerp(GroundColor, SkyColor,
               (worldNormal.y + 1)/2)*MaterialColor;
}
```

Listing 1.1: Hemispheric lighting in Cg

This shader program is a vertex shader. It computes the vertex normal in world-space `worldNormal` by using the inverse transpose of the model-world transform `ModelWorldIT`. The amount of incident light of the vertex is then computed by linear interpolation lerp of the sky and ground color where the weighting factor is determined the y-component world-space normal. The incident light is weighted by the material properties of the object. Finally, as required by all vertex programs the clip-space coordinates `clipPosition` are computed using the projection matrix `ModelViewProj`.

The parameters to the program which are marked as `uniform` are those which are constant for the duration of the shader program, whereas the other parameters vary over the vertices of the mesh. The extra field (`POSITION`, `NORMAL`, and `COLOR` in this example), known as the *semantic* of the parameter specify how they are mapped to application data. For example, an `in` parameter with semantic `NORMAL` is specified using OpenGL's `glNormal*` calls from the application.

Shaders require application programmers to write support code for every shader to be used. In order to access shader program parameters an application level identifier is needed. Accessing the parameter identifiers of our example shader from C++ would be as follows.

```
  cg_mvp = cgGetNamedParameter(cg_prog,"ModelViewProj");
```

```
cg_mvit = cgGetNamedParameter(cg_prog, "ModelViewIT");
cg_wv = cgGetNamedParameter(cg_prog, "WorldView");
cg_materialColor = cgGetNamedParameter(cg_prog,
                                        "MaterialColor");
cg_groundColor = cgGetNamedParameter(cg_prog,
                                      "GroundColor");
cg_skyColor = cgGetNamedParameter(cg_prog, "SkyColor");
```

Listing 1.2: Finding parameter identifiers

Each time the shader is used it must be bound after which each parameter has to be set to its current value using the corresponding Cg parameter identifier. The target for which the shader program has been compiled, called the profile of the program, must also be enabled.

```
cgGLBindProgram(cg_prog);

cgGLSetStateMatrixParameter(cg_mvp,
    CG_GL_MODELVIEW_MATRIX, CG_GL_MATRIX_IDENTITY);
cgGLSetStateMatrixParameter(cg_mvit,
    CG_GL_MODELVIEW_MATRIX,
    CG_GL_MATRIX_INVERSE_TRANSPOSE);
cgGLSetMatrixParameterfr(cg_wv,
    camera->inverseTransform());
cgGLSetParameter3fv(cg_materialColor, MaterialColor);
cgGLSetParameter3fv(cg_groundColor, GroundColor);
cgGLSetParameter3fv(cg_skyColor, SkyColor);

cgGLEnableProfile(cg_profile);
```

Listing 1.3: Binding shader program and setting parameters

Changing the parameters of the effect at run-time amounts to changing the local variables used here, for example `MaterialColor`, `SkyColor`, and `GroundColor`.

This code can be compiled and delivered together with the shader code as a complete package which can be used by the artist. However, there are limitations with this approach. All but the most trivial shaders require support code for setting parameters and renderer pipeline states. This support code is specific to each application due to differences in how textures are loaded and accessed, renderer pipeline state are set, *etc*. This gives unwanted dependencies between shaders and applications. Encapsulating these dependencies is a difficult problem since different applications have very different notions of what is important, for instance an artist's tool must be able to provide GUI components for manipulating the shader whereas an engine is primarily concerned with efficiency.

This encapsulation is made even more difficult when using shaders written by an external party. Externally written shaders use different interfaces but must still be accessible in the same manner as in-house developed ones in order to provide a unified working model for both artists and developers. The amount of work needed to adapt such shaders can often be too large.

1.3 Effects

The problems associated with using shaders as shown above are caused by a lack of encapsulation. Information associated with the shader and necessary for the shader to work is mixed with application code and not packaged together with the shader itself. This has called for a new level of abstraction and a new kind of entity to do the encapsulation. These entities are known as *effects*.

Today there are two major effect frameworks in use, the DirectX Effects by Microsoft [4] and CgFX by NVIDIA [2]. Both provide a text-based format where shader code, parameters and pass specifications are written in one file. This file is loaded by the application and compiled for the current run-time platform. The two formats are very similar and can in many instances be used interchangeably. Using CgFX the hemispheric lighting example can be implemented as:

```
float3 MaterialColor = { 1.0, 1.0, 1.0 };
float3 SkyColor = { 0.5, 0.5, 1.0 };
float3 GroundColor = { 0.0, 0.1, 0.0 };

float4x4 ModelViewProj : MODELVIEWPROJ;
float4x4 ModelViewIT   : MODELVIEWIT;
float4x4 WorldView     : WORLDVIEW;

shader code as in listing 1.1

technique Hemispheric {
  pass p0 {
    VertexShader = compile vs_1_1 main(ModelViewProj,
                                       ModelViewIT,
                                       WorldView,
                                       SkyColor,
                                       GroundColor);
  }
}
```

Listing 1.4: Hemispheric lighting in CgFX

This effect declares three parameters which are intended to be set at design time, `Material Color`, `SkyColor`, and `GroundColor`, and three parameters that are intended to be set at run-time by the application: `ModelViewProj`, `ModelViewIT`, and `WorldView`. The design-time parameters, also known as *tweakables*, may have associated annotations which can be used by design tools to automatically provide a suitable user interface for setting the parameter. For example a color picker control may be used to set the value of a color parameter. Run-time parameters on the other may have *semantic* identifiers associated with them, and similar to shader semantics, their purpose is to specify the mapping to application data without relying on parameter name. Instead an application can define a number of semantic identifiers which may be used in the effect.

Following the declaration of the effect parameters is the shader code. It is identical to Listing 1.1 and is therefore omitted here. Finally the effect declares a so called *technique* which describes number of rendering passes needed and the render states to be used in each pass. In this case there is a single rendering pass and in that pass the vertex shader `main` is to be compiled for the shader profile `vs_1_1` and the uniform shader parameters should have the values of the corresponding effect parameters.

Once loaded an effect can be used in the application like this

```
unsigned int numPasses;
effect->Begin(&numPasses, 0);
for (unsigned int p = 0; p < numPasses; p++)
{
  effect->Pass(p);
  renderMesh(mesh);
}
effect->End();
```

Listing 1.5: Effect usage with CgFX

The textures used by an effect are generally declared as tweakables where an annotation is used to specify the filename.

```
texture colorTexture : DiffuseMap <
   string File = "default_color.dds";
>;
```

Using a texture in a shader program is done indirectly through something called a *sampler* which specifies how the texture is accessed. Declaring a simple 2-dimensional sampler using linear minification and magnification filters for the above texture we write

```
sampler2D colorSampler = sampler_state {
  Texture = <colorTexture>;
  MinFilter = Linear;
  MagFilter = Linear;
};
```

This sampler is then passed to a shader program just as any other parameter.

Effects provide a number of mechanisms for separating applications from shaders. First, the effect format give a clear, high-level, and concise specification of shader programs, textures, and render states. This includes a unified method for handling multipass effects as well as having multiple implementations (fixed-function fall backs *etc.*) of the same visual effect. This specification is independent of the target architecture on which the effect is to run.

Second, tweakables provide the artist with a method for setting parameters at design time. This reduces support code since the engine only needs to concern itself with providing run-time parameters such as projection matrices *etc*.

Third, user-defined semantics provides a method for the engine to provide such run-time parameters. The application defines a number of semantic identifiers which it support and this creates an rudimentary interface to effects which, together with default values for parameters, relieves the effect developers of writing per-effect support code (cf. Listings 1.2 and 1.3).

However, as in the case with using shaders directly, there still exist a problem of encapsulation. The application defines an interface for the effects by using user-defined semantics. This interface is fixed, and this limits the number of shaders that may be expressed and used within a single application.

2 PyFX

The limitations in encapsulation of existing effect frameworks is due to the fact that effects are passive entities, text files, which are operated on by the application, which is the active party. If this relationship could be reversed so that effects are active instead, a better interface can be built where they can be responsible for retrieving the data they need from applications rather than the other way around. To achieve this reversed flow of control the effects must be embedded in a context which can do actual execution on their behalf.

2.1 PyFX overview

We have used Python, an existing scripting language to develop an active effect framework called PyFX. The current implementation supports applications using OpenGL and shaders written in Cg and its feature set closely resembles that of CgFX. In PyFX however, Python is used both to implement the framework and to write the effects themselves.

In an object-oriented language, it is natural to represent different effects as subclasses to a common effect base class. The subclasses implement specific functionality whereas functionality common to all effects are inherited from the base class. The object-oriented model also provides a natural mapping to the collaborative work-flow between programmers and artists. Effect programmers write new effects by making new effect subclasses, whereas the artist provides textures, sets parameters, etc. to make effect instances from existing classes.

Below is the hemispheric lighting example written in PyFX. It shows the Python class `Hemispheric` as a subclass of the general `Effect` class.

```
class Hemispheric(Effect):
   vs = Cg("""
      shader code as in listing 1.1
   """)

   SkyColor    = (0.5, 0.5, 1.0)
   GroundColor = (0.0, 0.3, 0.0)

   def __init__(self,
               MaterialColor = (1.0, 1.0, 1.0)):
      Effect.__init__(self)

      self.MaterialColor = MaterialColor

      self.technique = [Pass(VertexShader = vs())]
```

Listing 1.6: Hemispheric lighting in PyFX

The declaration has two main parts: the class variables and the constructor (the `__init__` member). The first class variable `vs` contains the shader program as a string wrapped by an instance of a Python class called `Cg`. and the other two class variables `SkyColor` and `GroundColor` are simply effect parameters. The class constructor, which creates new instances of the class, takes one additional effect parameter `MaterialColor` as an argument. The ability to differentiate between class variables and instance variables allows the effect writer to indicate that some parameters are intended to be the same for all instances of the class whereas other parameters may be different. The constructor body calls the superclass constructor and sets the instance variable `MaterialColor` of the object. Finally, the instance variable `technique` is set to specify that this is a single pass effect and that the pass should use the shader `vs` as its vertex shader.

Having instantiated this effect, for example like this

```
effect = Hemispheric(MaterialColor = (0.0,0.0,1.0))
```

it can be applied to a mesh by

```
while effect.hasMorePasses(mesh):
    renderMesh(mesh)
```

The `Effect` member function `hasMorePasses` does setup for each pass of the effect and also specifies how many times the mesh needs to be rendered.

Having applied effects to meshes the next issue is the passing of information from the application to the effect and its shader. In PyFX this data can be passed through a number of different channels.

The most obvious way is through constructor parameters when the Hemispheric effect is instantiated. The example above shows how `MaterialColor` is set to the color blue.

In the hemispherical lighting example the constructor parameters correspond exactly to an instance variable of the effect. Another method of passing data to the shader is to assign new values to this variable. For example

```
effect.MaterialColor = (0.5, 0.5, 1.0)
```

changes material color so that it is now light blue. Similarly class variables can also be assigned new values

```
Hemispheric.GroundColor = (0, 0, 0)
```

The framework then make sure that these changes are made available to the shader code.

Another type of parameters are the transformation matrices used by the effect; `ModelViewProj`, `ModelViewIT` and `ViewWorld`. The matrices `ModelViewProj` and `ModelViewIT` can be retrieved from the OpenGL rendering pipeline and in PyFX this is handled automatically.

The third parameter `ViewWorld` needs special treatment. It is the inverse camera transform, used by the effect to compute the `ModelWorld` transform, neither of which can be automatically retrieved from the pipeline. It must therefore be provided by the application. Since this parameter is the same for different instances it makes sense to make it a class variable. However, it is even more general than that since you could easily think of other effects that might need it. In this case we can therefore set it as a class variable on the `Effect` base class, for example:

```
Effect.ViewWorld = camera.inverseTransform()
```

Yet another method for passing data from the application is when the effect needs additional data at each vertex, *i.e.* non-standard varying parameters. In our hemispherical lighting example this is not the case, but a more advanced version of hemispheric lighting can used to illustrate this case [9]. This version use additional per-vertex mesh data, called the *occlusion factor*, which determine the amount of hemispheric light which reach the point in question. If the shader program has the following prototype

```
void main(..., float OcclusionFactor : COLOR )
```

Then, if the mesh has an array member `OcclusionFactor`, PyFX will automatically bind this to the varying parameter with the same name.

2.2 PyFX details

Techniques

The structure of PyFX effects is inspired by that of DirectX Effects and CgFX frameworks. As in these each effect contain one or more techniques. Each technique contain a number of passes which are to be run consecutively. Each render pass has associated render states specifying the necessary pipeline states required to run the pass. Specifying that back-face culling should be disabled while alpha-blending is enabled is written in CgFX as

```
pass p0 {
  CullMode = NONE,
  AlphaBlendEnable = True
}
```

In PyFX the same render pass specification would look like

```
Render(CullMode = None,
       AlphaBlendEnable = True)
```

A single technique effect for CgFX is shown in listing 1.4. The corresponding effect in PyFX is given in listing 1.6. Providing two techniques `Hemispheric` and `Ambient` in CgFX is done by providing multiple technique blocks

```
technique Hemispheric {
  pass p0 {
    VertexShader = compile vs_1_1 main();
  }
}

technique Ambient {
    pass p0 {
      Color = <ambientColor>;
    }
  }
```

The same would be written in PyFX as

```
technique = {}
technique['Hemispheric'] = [
    Render(VertexShader = vs())]
technique['Ambient'] = [
    Render(Ambient = AmbientColor)]
```

Textures

Texturing in PyFX is, as in CgFX, divided into textures and samplers. Declaring the same texture and sampler as above (Section 1.3) in PyFX would be written as

```
colorTexture = Texture(filename="default_color.dds")
colorSampler = Sampler(colorTexture,
                       MinFilter = Linear,
                       MagFilter = Linear)
```

This sampler can then be used either by a shader program, using parameter resolution, or in the fixed-function pipeline by

```
Render(Texture0 = colorSampler)
```

Multi-texturing is naturally supported and when using multiple textures in a shader program this is automatically handled by the shader parameter resolution code. For fixed-function effects the different texturing-units are accessible via

```
Render(Texture0 = colorSampler,
       Texture1 = lightMapSampler)
```

where `colorSampler` and `lightMapSampler` are two samplers with appropriate settings.

Shaders

Shaders are provided via strings wrapped with classes providing information on the type of shader code contained in the string. Sometimes it is useful to specify the target for which a given shader should be compiled. This can be achieved via

```
Render(VertexShader = vs(target=arbvp1))
```

Also, passing explicit parameters to shader programs can be done by adding keyword arguments to the shader invocation. Suppose we have an outlining effect which draws a gradually more transparent outline around an object. This effect should run multiple passes with the same shader program (called `outline`) but with a parameter `offset` determining the size and opacity of the outline

```
[Render(VertexShader = outline(offset=1.0)),
 Render(VertexShader = outline(offset=0.75)),
 Render(VertexShader = outline(offset=0.5)),
 Render(VertexShader = outline(offset=0.25))]
```

Listing 1.7: Setting compile-time parameters

The same thing can be expressed in CgFX but the result is more verbose since every shader parameter must be passed explicitly.

If a shader program source code `shader` contains multiple programs, say a vertex shader `shadeVertex` and a pixel shader `shadePixel`, these programs entries can be accessed by the corresponding methods on the shader object

```
Render(VertexShader = shader.shadeVertex(),
       PixelShader = shader.shadePixel())
```

Parameter resolution in PyFX

Application level variables having the same name as the shader program parameters are used as arguments to the shader program. These arguments are defined in one of the following places:

- Either it is a compile-time parameter to the shader program (see listing 1.7), or
- an attribute of the effect object, or
- an attribute on the mesh currently being rendered, or, lastly,
- a member of a predefined set of state parameters giving access to current pipeline states.

Attributes of the effect instance include both instance parameters, such as the material color parameters above (Section 2.1), and class parameters, `SkyColor` and `Ground Color` above. As usual the class scope includes the scope of its superclass making the `WorldView` transform accessible to the shader programs. In the above examples the `OcclusionFactor` is a mesh attribute and them `ModelViewProj` and `ModelViewIT` matrices are both pipeline state parameters.

If there are multiple variables with the same name the order of precedence is that compile-time parameters take precedence over instance attributes, which take precedence over class variables. Effect class variables take precedence over mesh attributes and state parameters are used last.

This gives a natural correspondence between parameters to the shader program and application data. Setting effect class-specific values amounts to setting effect class-variables whereas effect instance-specific values are set by setting the appropriate attribute on the effect instance in question. Effects take a more active role since they are allowed to extract information from the mesh currently being rendered thus minimizing the amount of application level dependencies.

The mapping is recursive so the following Cg shader program

```
struct Light {
  float3 position;
  float4 color;
};

void main(..., uniform Light light) { ... }
```

will use `position` and `color` member of the application level variable `light`.

Name maps

The lookup scheme above gives great flexibility in both writing and using effects. However when dealing, for instance, with third-party effects a name-based lookup is not always sufficient since naming conventions may differ. Suppose we wish to use an effect which uses the name `DiffuseMap` where our application use `DiffuseTex`. An obviously unattractive solution would be to add `DiffuseMap` to our code and make sure to update it each time we change `DiffuseTex`.

PyFX solves this problem by having user defined *name maps*. The `Effect` class allows us to pass a dictionary of how parameter names at the shader level should be mapped to parameter names at the application level. Defining a dictionary containing our mappings and passing it to the effect nicely handles this.

```
myNameMap = {'DiffuseMap' : 'DiffuseTex'}

effect = SomeTexture(nameMap = myNameMap)
```

A request for the `DiffuseMap` will now be automatically translated to a requests for `DiffuseTex`.

Language embedding

The fact that Python is used to write effects and not only for implementing the framework is convenient but not strictly necessary. It would have been possible to write an interpreter and for example use the CgFX format. However, the complete *embedded* of effects in Python has the advantage that all the ordinary language features such as lists, tuples, loops, functions, dictionaries, list comprehension, etc. are available to the effect writer[11]. As a very simple example we could have used a list comprehension to write the pass specification of the outline effect (listing 1.7) as

```
[Render(VertexShader=outline(factor=f))
                        for f in [1.0, 0.75, 0.5, 0.25]]
```

Module-style effects

When using concrete subclasses of `Effect` the application needs to know about every such class at compile-time, something known as the *library problem*. This is clearly not desirable in a graphics application and it was one of the problems effects where created to alleviate. In traditional object-oriented design it is solved by introducing an abstract factory for handling instantiation of concrete subclasses [6]. However, using the flexibility of Python we can provide a method which simultaneous solves this problem while giving a cleaner and more direct syntax for declaring effects. Effects can be implemented simply as Python modules which can be loaded by

```
effect = Effect('Hemispheric')
```

This loads the `Hemispheric` effect module which can be used just as any other effect. Note however, that since the actual subclass is not known setting class variables such as `GroundColor` (cf. Section 2.1) is not possible.

Image processing

PyFX also provides a mechanism for specifying render targets other than the frame buffer to which rasterization should occur. Furthermore it is possible to have passes which do not render geometry but instead do shader based image processing. These two features, which are not available in CgFX or DirectX, allow effects such as blurring, edge detection, image compositing *etc.* to be expressed in an application independent manner.

3 Implementation

PyFX is implemented on top of PyOpenGL [20] and a SWIG [21] generated interface to the Cg runtime library. The implementation consists of about 800 lines of Python code. The bulk of it is concerned with basic functionality needed in any effect framework such as loading and binding textures, compiling shader programs, and initializing OpenGL extensions. The remaining part implements the distinguishing features of PyFX, *i.e.* mapping declarative state specification to function invocations and performing parameter resolution. This part is remarkably small, only about 10% or 80 lines of code. This compactness is possible because of Python's dynamic object model and introspection facilities.

3.1 Renderer management

The entry point of the PyFX framework is provided by the top-level `Effect` class. It is essentially a container for other objects, *i.e.* techniques, passes, textures, samplers, and shaders. These classes interact with the underlying graphics API through a global `RenderState` singleton class which implements manipulation of the renderer pipeline state. The majority of its methods correspond one-to-one to the available state variables. For instance the `CullMode` state is implemented as

```
class RenderState:
  ...
  def CullMode(self, val):
      if val:
          glEnable(GL_CULL_FACE)
```

```
            glFrontFace(val)
        else:
            glDisable(GL_CULL_FACE)
```

When a `Render` is activated it instructs the `RenderState` object `state` to change the state of the rendering pipeline. This is done by mapping every state specified in the pass object to a method invocation. For example, a pass specified by

```
Render(Color = (1.0, 0.0, 0.0),
       CullMode = None)
```

will result in the following method calls on:

```
state.Color((1.0, 0.0, 0.0))
state.CullMode(None)
```

Doing this mapping is the responsibility of the `Render` class and by using the dynamic introspective features in Python, it can have a very small implementation:

```
class Render:
  def __init__(self, **kwords):
      self.kwords = kwords

  def use(self, state):
      for s,v in self.kwords.items():
          marshalFX(state,s,v)
```

The `marshalFX` maps the state name `s` to the proper method name and calls this method with argument `v`. It is similar to the marshaling used by RPC (remote procedure calls), whereby serialized data (dictionary tuples) are converted to method invocations. Implementing `marshalFX` is a two-liner:

```
def marshalFX(obj, name, *args):
    method = getattr(obj,name)
    return method(*args)
```

3.2 Texture and sampler state

The class `Texture` provides an encapsulation similar to `RenderState` but for the available texture states such as filtering, texture coordinate wrapping, etc. The texture state information is maintained by the corresponding `Sampler` and it is responsible for marshaling this information to method invocations on the `Texture` object.

When a `Sampler` is used by either the fixed-function pipeline or by a shader the framework allocates a free texture unit and asks the sampler to bind itself to that unit.

3.3 Shaders

Just as samplers are responsible for performing binding textures and setting texture states, every `Shader` object is responsible for performing its own loading, binding, compilation, and parameter resolution. This implementation is actually contained in subclasses for different shader programming languages. Currently the only subclass implemented is `Cg`.

When the pass specifies a vertex or fragment shader the `state` object instructs the shader to bind itself. A shader binding itself includes setting the value of every parameter needed by the shader. The mapping scheme of PyFX between parameters and application variables is implemented by a `Resolver` object whose responsibility it is to search the effect and mesh name spaces as well as providing name mapping (Section 2.2). The resolver searches a list of objects for a given attribute, optionally transform the attribute name via the name mapping dictionary:

```
class Resolver:
   def __init__(self,nameMap, *objs):
       self.nameMap = nameMap
       self.objs = objs

   def __getattr__(self,attr):
       if self.nameMap.has_key(attr):
          attr = self.nameMap[attr]

       for obj in self.objs:
          if hasattr(obj, attr):
              return getattr(obj, attr)
```

The `Cg` class use the resolver to locate shader parameters and set these by invoking the corresponding CgGL functions. For simple variables the `cgGLSetParameter`-family of functions are used. Aggregate parameters, such as arrays and structs, are handled by iterating over the members and setting each element recursively.

4 Conclusions and future work

The most prominent features provided by the PyFX framework is the decoupling of effects from the application. This "activation" of an effect, enabling it to obtain needed data from *e.g.* the current mesh without the need to introduce application level support code, greatly reduces dependencies between effects and the application. Using this activation together with the introspection features of Python gives a natural mirroring between data at the application level and data at the level of shader programs. This also eliminates the need for user-defined semantics since there is no longer any need to provide *ad hoc* hooks for applications to provide specialized data and operations. Instead the object-oriented extensible nature of the host programming languages can be used to provide this functionality natively at the effect level.

There are some limitations however, in the current implementation of PyFX. Support for manipulating fixed-function effect parameters is limited. Consider a simple effect such as

```
class SimpleColor:
    color = (1,0,0)
    technique = [Render(Color=color)]
```

Manipulating the `color` attribute of this effect will not have the desired effect, the color used for drawing will remain red. The reason why the parameter resolution algorithm (Section 2.2) can not be applied in this case is that it requires access to the parameter names. These names are only available to shader based effects where they are supplied by the Cg run-time library.

The overall purpose of PyFX is to be a flexible tool for investigating what kind of features and functions are needed to make effect programming as easy and productive as possible. Future work includes investigating how effects can be combined efficiently at run-time allowing, for instance, stencil-buffer shadow algorithms to coexist with other visual effects.

Acknowledgement

The authors wish to thank Benny Åkesson and Andreas Hansson for help with implementing the framework as well as providing a number of interesting effects testing various aspects of PyFX. We also extend our thank to Chuck Mason for providing the initial Python interface to Cg.

Bibliography

[1] Paul Jaquays amd Brian Hook. *Quake III Arena Shader Manual.* Id Software Inc., 12th edition, December 1999.

[2] CgFX 1.2 Overview. `http://developer.nvidia.com/`.

[3] Robert L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press, 1984.

[4] DirectX SDK Documentation. `http://msdn.microsoft.com/`.

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

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

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

[8] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298. ACM Press, 1990.

[9] Hemispheric lighting. Example in DirectX 9 SDK documentation. MSDN Library, `http://msdn.microsoft.com`.

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

[11] Calle Lejdfors and Lennart Ohlsson. A scripting tool for real-time effect programming. In Vaclav Skala, editor, *WSCG' 2005*, pages 37–38, 2005.

[12] 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.

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

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

[15] Steven Molnar, John Eyles, and John Poulton. Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 231–240. ACM Press, 1992.

[16] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168. ACM Press, 1998.

[17] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM Press/Addison-Wesley Publishing Co., 2000.

[18] Ken Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.

[19] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 159–170. ACM Press, 2001.

[20] PyOpenGL project. `http://pyopengl.sf.net/`.

[21] SWIG project. `http://www.swig.org/`.

Paper II

PyFX: A framework for real-time graphics effects

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

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

Abstract

Programming real-time effects for contemporary GPUs requires writing shader programs to run on the GPU as well as code for the render state setup logic performed by the CPU. While the GPU parts are well supported by high level programming languages, the effect frameworks commonly used for the CPU parts are lacking both in functionality and expressive power, which makes them difficult to work with.

In this paper we present an effect framework implemented as an embedded language in Python. We show that this high-level language for effect descriptions provide increased expressivity, without sacrificing declarativity of other frameworks. We show how some additional functional features, image-processing and off-screen render targets, cooperate with the effect language giving a rich environment for experimenting with both functional and expressive features of effect programming.

Technical report, Lund University, LU-CS-TR:2005-233

1 Introduction

Special effects in cinematic graphics have long relied on procedural techniques and in the last few years the evolution of graphics processors have made these techniques available for use in real-time graphics as well. Programming real-time effects is today significantly harder than programming cinematic effects. The reason is partly that the real-time constraints makes the problem harder simply because limits on the execution time implies restrictions on the algorithms that can be used. Another reason is that the tools and techniques available are not yet as mature and this makes the development process harder than it has to be.

In the world of cinematic graphics programming visual effects is known as shader programming. The term shader programming is also used in the context of real-time effects, but here it is commonly used to refer only to the part of the effect which is executed as a program on the GPU. We therefore use the term *effect programming* for the process of creating the complete real-time effect, including the shaders for the GPU but also code to be run on the CPU.

Real-time shaders can today be programmed in a number of high-level languages. NVIDIA's Cg [16], Microsoft's HLSL [7] and the OpenGL Shading language, GLSL, [12] are all based on the Renderman Shading Language, the established standard for programming cinematic effects. They have come a long way towards being a flexible and efficient development tool for the GPU parts of an effect. Programming the CPU part of the effects, however, has still very limited support. Loading shaders to the GPU, binding their run-time parameters, setting pipeline states and controlling the execution of multiple passes are commonly done in application specific code and not integrated with the rest of the definition of the effect.

The current approach to effect programming is the use of *effect frameworks*. An effect framework handles pipeline state manipulation including downloading shader programs and textures, doing parameter passing from the application to the shader program, and for orchestrating multiple passes of an effect. It provides some facility for loading effects and shader programs, typically based on a text file format in which shader programs, pipeline states and pass specifications are listed.

The effect framework idea was first used in the Quake 3 engine [1] to allow user scripting shaders, or what we would call effects, to control visual appearance of in-game characters and objects. The Q3 framework predates widely available programmable shader hardware and therefore lack many of the features of current effect frameworks. The two effect frameworks which are the most widely used are DirectX FX and CgFX which are extensions of the shader programming languages HLSL and Cg, respectively. Both these frameworks provide the ability to specify real-time effects using multiple shader programs and passes using a special file format syntax.

Although current effect frameworks improve the encapsulation of the GPU part and the CPU part of an effect, they still suffer at least two major drawbacks. The most critical one is that they lack some important features such as render to texture and image processing which are necessary for writing many of the effects used in modern graphics applications. Second, the syntax of current frameworks is rather restricted, *e.g.* there is no support for expressing abstraction or repetition, which many times can make the writing of effects tedious and error prone. Limited facilities for sharing common parts of effects result in redundancy and code duplication.

In this paper we present PyFX, an effect framework based on the Python [23] programming language. We show how an effect framework can be embedded in a very high level general purpose language enable easy development of new extension, in particular off-screen rendering and image processing. Furthermore embedding allows language constructs currently not supported by DXFX or CgFX, such as function definitions, classes, conditionals, and loops, to be used in the construction of effects. This enables design techniques established for other kinds of software to be applied to effect programming as well.

This paper is organized as follows. In Section 2 we present some related work. Section 3 gives an overview of current effect frameworks. In Section 4 we present PyFX followed by some examples (Section 5) which emphasize the advantages of PyFX. Finally we conclude with a discussion (Section 6).

2 Related work

Shader programming was started in 1984 when Cook introduced shade trees [4]. This represented a move away from the fixed function nature of previous systems to an interpreted model giving much greater flexibility for writing visual effects. In 1990 this gave rise to the RenderMan [8] shading language which became an industry standard for writing off-line shaders. Several propositions on how the power and flexibility of off-line shader programming systems can be transferred to the world of real-time graphics have since been put forward.

One direction is taken by Olano and Lastra [20], who describe a RenderMan-like real-time shading language for the PixelFlow system [19]. This system consists of a SIMD array of general purpose processors for which shader programs are compiled via C++ and executed. While this system is well-suited for writing real-time shaders it bears little resemblance to the architecture of current GPUs.

Peercy *et al.* [21], present a system for compiling RenderMan programs to multipass rendering on using an OpenGL 1.2 implementation extended with imaging support, high-precision data types (16 bit floating point), and dependent texturing. The key realization is that the graphics pipeline can be used as a SIMD processor where different OpenGL states correspond to different SIMD instructions operating in parallel on a set of fragments. A restricted version, called ISL, of the RenderMan language, which can run on top of any OpenGL 1.2 implementation, is also presented.

The realization of Proudfoot *et al.* [22] that shading computations are carried out at different frequencies lead to a language which maps well to present day GPUs. The computational frequencies isolated where constant, per-group, per-vertex, and per-fragment. The compiler can use frequency information of a computation in order to map it to a particular stage of programmable pipeline.

Today a number of shader languages have found widespread use in the industry. These are HLSL [7] by Microsoft and Cg [16] by NVIDIA which are both very similar. The OpenGL Shading Language [12] achieved ARB approval in 2003 and is included in the OpenGL 2.0 specifications [24]. All these languages are based around explicit separation of per-vertex and per-fragment computations (cf. Proudfoot *et al.*) and follow the uniform and varying data classification introduced in RenderMan.

On the consumer side, games such as Quake3 by ID Software makes heavy use of multi-pass multi-texture algorithms. The Q3 shader [1] format (here called an *effect* format) provides a specialized language for controlling blending state, texture generation, fogging and texture application mode of multiple textures. The engine can then use, if available, multiple texture units to reduce the number of texture application passes needed. The format marks a first step in effect programming but it does not provide enough control be useful in a more general context. In particular, since Quake3 shaders predates consumer-level programmable graphics hardware, it does not support vertex or pixel shader programs.

Following in the footsteps of Quake3's shader format Microsoft introduced, coincident with the first generation of programmable hardware, the DirectX Effects [5] (abbreviated here as DXFX). DXFX provide a large superset of the interface introduced by Q3 shaders allowing for vertex and pixel shader programs, stencil-, alpha-, and depth buffer operations, multiple passes, *etc.* to be used in the description of a visual effect. CgFX [3] was introduced together with the Cg language and provide an implementation of DXFX for a larger variety of platforms.

A related approach was taken by Lalonde and Schenk [13] with the EAGL framework. This framework provides a portable method of describing the association of render methods (shader programs and render state setup annotated data binding semantics) and art assets, typically triangle meshes, allowing for efficient rendering on a number of platforms (PC/XBOX, Playstation 2, and GameCube). An off-line compiler takes combinations of render methods and art assets and generates platform specific representations which can be efficiently rendered by the runtime system. Contrary to current effect frameworks, EAGL does not provide support for writing cross-platform render methods, every render method used must be reimplemented for every platform used.

The Sh shading language [17] demonstrates that a shading language can be implemented as embedded language in C++. This embedding gives the shader developer access to high-level language features, such as classes, templates, functions, and user-defined types, to be used in the construction of shader programs. Sh also provide support for run-time construction and composition of shader programs [18]. However, although it is implemented in C++ it lacks facilities for abstracting and expressing the CPU parts of effects.

The Vertigo shading language [6] approaches shader programming from a novel angle. It is implemented as an embedded language in Haskell [10] and uses pure functions, *i.e.* functions without state or side-effects, to model the stream-like nature of the GPU. This results in a clean high-level model for programming shaders for generative geometry.

3 Current effect frameworks

Current effects frameworks such as DXFX and CgFX provide a number of features which simplify programming real-time visual effects. DXFX and CgFX extend HLSL and Cg, respectively, with the ability to

- declare effect variables which can be either user-editable, or *tweakable*, for data such as textures or colors, or engine internal, so called *non-tweakables*, for engine specific data such as transformation matrices.
- declare different implementations of an effect suitable for different platforms. Each of these so called *techniques* list a number of passes with each pass containing the render pipeline states to set before requesting the application to submit geometry to the render pipeline.

Both frameworks rely on effect specifications which are stored in text files and loaded at run-time. The effect file lists the variables, shader programs, techniques and passes making out the effect. Effect variables can optionally be annotated with application specific data such as the valid range of a parameter or the default filename of a texture to facilitate integration with, for instance, GUI development tools. Variables may also have an associated semantic, consisting of a string identifier, which can be used by the application to provide data independent of variable name, providing an abstraction when passing data to the effect.

Effects, being implemented in terms of external text files, can be changed without requiring recompiling the application. Using a standard set of annotations and semantics, introduced in DirectX 9, the format provides an application independent mapping of application data to effect data.

However, the file format is closed and the frameworks can not easily be extended with new functionality. Syntactic-wise the effect format allow the use of C-style preprocessor for writing simple syntactical extensions. Together with the possibility to group states in so called state blocks this provides a basic form of abstraction. On the downside, the preprocessor based approach lack the most basic abstraction and data-hiding functionality making it difficult to use effectively. Furthermore functionality such as looping and conditionals are also missing, requiring generative effects and effects containing hardware-specific implementations to rely on effect-specific application level code. For a more in-depth review of the problems associated with current effect frameworks we refer to [14].

4 PyFX

In this section we present our effect framework, PyFX, which is implemented as an embedded language in Python. Its main purpose is to be a tool for investigating which features and characteristics that are useful and desirable for effect programming. The fact that PyFX is embedded in a fully fledged programming language immediately makes it easier to write effects since it allows the use of all the language features from the host language. Using function definitions, loops, conditionals and modules to express and share common parts, the description of an effect becomes shorter and more clear.

The features in PyFX include those found in the DXFX and CgFX frameworks and in addition it provides:

- *Render-to-texture* – The framework can render to off-screen area which can be used as a texture in later stages of the effect or by another effect entirely.
- *Image processing support* – GPU based image processing operations can be applied to any texture or off-screen area.
- *Support for shader interfaces* – PyFX enables easy use of Cg's interfaces allowing run-time construction and composition of shader programs.

PyFX is built on top of OpenGL. It is designed to be independent of shader language and it currently supports Cg and GLSL. The implementation and application level interface of PyFX is described in more detail in [14].

4.1 Overview

The basic building block in an effect in PyFX is a "processing step" which is a generalization of the notion of a pass in other effect frameworks. Each step may or may not require the application to send geometry to the GPU. Currently there are two types of processing steps:

- *RenderGeometry* – these are the usual pass of other effect frameworks. Sets up the appropriate states and then instructs the application to transmit geometry.
- *ProcessImage* – used to perform 2D image processing between two images (which may reside in either textures, off-screen areas or the current screen buffer). It supports floating point target and source images/buffers allowing HDR image processing.

In addition to these functional features, the framework also provide, through language embedding, a complete programming language in which effects can be expressed. This has several benefits for effect programmers since it enables the use of common software design methodologies, such as abstraction and sharing in the construction of an effect. This allows the effect writer to express an effect in a clear, to-the-point manner making development and debugging easier.

Every aspect of the framework is implemented as a class allowing easy extension and specialization. Together with using embedding in a high-level language, this enables engine and framework writers to experiment with new features with minimal impact on the rest of the framework.

5 Examples

As an example of the features common to PyFX, DXFX, and CgFX we will present an effect for doing bump-mapping using a normal map [2]. While PyFX supports both Cg and GLSL only Cg will be used for example code.

The bump mapping effect uses a vertex program to translate the surface normal to tangent space. The fragment program then manipulates the normal using a normal map and a scaling parameter. The resulting normal is then used for shading computations. The initial part of the fragment shader program is shown below:

```
float4 fs(float2 texcoord : TEXCOORD0,
          //normal in tangential frame
          float3 normalT  : TEXCOORD1,
          ..., // parameters needed for lighting
          uniform sampler2D NormalMap,
          uniform float Scale) : COLOR
{
  float3 dN = tex2D(NormalMap, texcoord);
  normalT += Scale*(dN*2-1)
  normalT = normalize(normalT);
  // compute and return color ...
}
```

Listing 2.1: Bumpmapping in Cg

In contrast to DXFX and CgFX where shader code is mixed with parameter declarations, PyFX uses wrapper functions and classes to indicate which parts of the effect are shader programs and which are parameters *etc.* The wrapper for a Cg shader program is called `Cg` and is used as follows.

```
bumpmap = Cg(""" code as in Listing 2.1 """)
```

The triple-quotes `"""` are used by Python for multi-line string literals. Next we list the parameters of the effect together with their default values.

```
Scale = 0.2
NormalMap = Texture("default_normalmap.png")
```

Now we are ready to define the technique of this effect. In PyFX it consists of a single render step which uses `bumpmap` vertex and fragment programs.

```
technique = [RenderGeometry(
                VertexShader = bumpmap.vs(),
                FragmentShader = bumpmap.fs())]
```

This effect does not use any of the special features of PyFX and we could just as well have written it in CgFX or DXFX. The result of doing so would have been more code since we have to explicitly declare every parameter, including non-tweakables, used by the shader programs. In PyFX common variables, such as transformation matrices *etc.*, are passed implicitly to the shader program. This is described in detail in [14].

5.1 Generative effects

Because of the language embedding in Python we can use, for instance, repetitive elements and function abstractions, when writing our effects. Consider an effect for rendering furry objects. Such an effect can be achieved by rendering the object surrounded by a number of shells where the transparency of each shell increases with the distance to the object. This gives the impression of fur with decreasing thickness with increasing distance from the object [15]. Assuming we have a `FurShellShader` program for rendering a single fur shell `shell` of the object, we can describe a step for rendering fur-shells by the following constructor function:

```
def RenderFurShell(s):
    shell = s/FurThickness
    return RenderGeometry(
                AlphaBlendEnable = True,
                SrcBlend = SRCALPHA,
                DestBlend = ONE,
                VertexShader = vs(Shell=shell),
                FragmentShader = fs(Shell=shell))
```

Drawing the complete furry object using `NumberOfShells` shells amounts to rendering the solid object followed by rendering each fur shell, which is done using the following code:

```
technique = [RenderGeometry()] + \
            [RenderFurShell(i)
                for i in range(1,NumberOfShells)]
```

The use of abstraction and high-level constructs allow us to describe the fur effect succinctly. Key parameters such as the number of shells used or fur thickness can easily be changed without modifying other parts of the effect.

The same fur effect expressed in CgFX or DXFX would be much longer and more difficult to read since those formats lack a notion of repetition. With each shell render step written explicitly it is, for example, more difficult to change the number of shells used.

5.2 Image processing

A simple widely-used example of image processing is the glow effect [9] used to simulate the nimbus due to atmospheric scattering which appear around brightly lit surfaces. It works by rendering an object to the screen, rendering the glowing parts of the object to an off-screen buffer, blurring the off-screen buffer and then additively blending the result to the screen. To express this in PyFX we start by introducing some helper functions for rendering the glow regions, blurring a buffer and additively blend some buffer onto some buffer.

```
def RenderGlowRegions(target):
    return RenderGeometry(
                Target=target,
                VertexShader=glowMask.vs(),
               FragmentShader=glowMask.fs())

def GaussianBlur(source):
    ...

def AdditiveBlend(source, target):
    return ProcessImage(Source=source,
                        Target=target,
                        SrcBlend = SRCALPHA,
                        DestBlend = ONE)
```

The technique which performs blurring can now be written simply as

```
technique = [RenderGeometry(),
             RenderGlowRegions(blurBuffer),
             GaussianBlur(blurBuffer),
             AdditiveBlend(blurBuffer, Screen)]
```

The result is a readable specification of what the effect does and how it does it. Writing this effect in DXFX or CgFX is currently not possible without using application-specific workarounds.

Sharing common code

If we have two different effects which both do blurring, it makes sense to factor out this common part and describe it only once. We can, for example, define a function `BlurPostProcess`, contained in a module `Blurring`, by

```
def BlurPostProcess(source,target):
    return [RenderGlowRegions(source),
            GaussianBlur(source),
            AdditiveBlend(source, target)]
```

Now our original glow effect can be implemented as

```
import Blurring

blurBuffer = ...

technique = [RenderGeometry()] + \
   Blurring.BlurPostProcess(blurBuffer, Screen)
```

The other effect is obtained by the replacing `technique` by some other step sequence followed by the blur post-processing operation.

```
technique = [...] + \
   Blurring.BlurPostProcess(blurBuffer, Screen)
```

Using Python's modules we can share code between multiple effects simplifying the construction of many effects.

5.3 Supporting shader interfaces

The Sh shader language provides support for combining shader programs at run-time. Cg provide a similar method through the use of so called *interfaces*, similar in concept to interfaces in the Java programming language [11].

Prior to the introduction of interfaces the programmer was required to write one shader program for every combination of material and light-source. This results in a combinatorial explosion of the number of shader programs needed in an application. Interfaces provide us with a mechanism for abstracting implementation of a part of a shader program from its usage pattern. As an example, we can use the following Cg interface for a light source.

```
interface LightI {
  void intensityAt(in float3 position,
                   out float3 lightDirection,
                   out float3 color);
};
```

An implementation of a light source must be able to, given a point, return the color and direction of the incident light at that point. A program can use the `LightI` interface as:

```
void main(in float3 position,
          in float3 normal,
          out float3 color,
          uniform LightI Light)
{
  float3 L, C;
  light.intensityAt(position, L, C);
  color = //compute color
}
```

Listing 2.2: Using interfaces in Cg

Using this in PyFX we wrap up the program above in a Cg wrapper as:

```
fs = Cg(""" code as in Listing 2.2 """);
```

We can implement a number of different light sources which support the above interface. For instance a non-attenuated point light source can be implemented as

```
struct PointLight : LightI {
  float3 Position;
  float3 Color;

  void intensityAt(in float3 position,
                   out float3 lightDirection,
                   out float3 color)
  {
    lightDirection = normalize(position - Position);
    color = Color;
  }
};
```

Following this general outline we can easily implement spotlights, lights with attenuation, cube-mapped lights *etc.* We put all the light definitions in a `Lights` module.

```
pointLight = CgIImpl(""" code as above """);
spotLight = CgIImpl(""" ... """)
cubeMappedLight = CgIImpl(""" ... """)
```

Interface implementations are wrapped in PyFX `CgIImpl`-wrappers to indicate that they are not complete programs, only implementations of interfaces which are not executable in their own right.

Using the interfaces works just as variables so using the above program with a point light located at $(0, 100, 0)$ having red color is just

```
myPointLight = pointLight(Position=(0,100,0),
                          Color=(1,0,0))

technique = [RenderGeometry(
                FragmentShader =
                   fs(Light = myPointLight), ...)]
```

Changing the light source amounts to changing a single line in the effect file. The framework also support changing the light source at run-time, allowing flexible shader program composition to be used as an integral part of an application.

6 Discussion and conclusions

We have presented an effect framework which improves on current frameworks in two respects. By being an embedded language it can freely utilize the features of its host language. In this respect PyFX is similar to Sh and Vertigo, although these systems have slightly different focus. However, when the language embedding is done a low level language like C or C++, the power of host language features come at the cost of sacrificing the declarative style of DirectX FX and CgFX. Due to the high level character of Python, PyFX is able to provide the best of both worlds. It is both declarative and has a rich set of language features.

Furthermore, PyFX provides support for render to texture and image processing, features which are needed to write many common effects but which are not supported by current frameworks. With access to the source code of an existing framework, these features would probably be fairly straightforward to implement, and it is even likely that they will appear in some future version. Our experience is however with a framework which is embedded in an flexible language such amendments can be added very easily. It is possible to have a very short turn-around time for adding or modifying a feature, get feedback from using it and then change it again. Since it is likely that the wish list for effect framework features will continue to grow for some time still, we believe that this agility makes PyFX a suitable platform for exploring and evaluating the design space of effect frameworks.

In summary, PyFX framework represents work in progress but it already provide a flexible environment for prototyping and experimenting with effects and effect frameworks alike. We hope to continue exploring methods for providing good programming environments for the border land of CPU/GPU interaction. Hopefully we will also be able to provide a solid ground for extending the framework to handle GPGPU algorithms as well as providing further functional additions.

Bibliography

[1] Paul Jaquays amd Brian Hook. *Quake III Arena Shader Manual.* Id Software Inc., 12th edition, December 1999.

[2] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292. ACM Press, 1978.

[3] CgFX 1.2 Overview. `http://developer.nvidia.com/`.

[4] Robert L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press, 1984.

[5] DirectX SDK Documentation. `http://msdn.microsoft.com/`.

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

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

[8] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298. ACM Press, 1990.

[9] Greg James and John O'Rorke. *GPU Gems*, chapter Real-Time Glow, page 816. Addison Wesley Professional, March 2004.

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

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

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

[13] Paul Lalonde and Eric Schenk. Shader-driven compilation of rendering assets. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 713–720. ACM Press, 2002.

[14] Calle Lejdfors and Lennart Ohlsson. Pyfx – an active effect framework. In Stefan Seipel, editor, *SIGRAD 2004. The Annual SIGRAD Conference. Special Theme – Environmental Visualization. November 24, 2004, Gävle, Sweden*, number 13 in Linköping Electronic Conference Proceedings, 2004.

[15] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 227–232. ACM Press, 2001.

[16] 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.

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

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

[19] Steven Molnar, John Eyles, and John Poulton. Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 231–240. ACM Press, 1992.

[20] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 159–168. ACM Press, 1998.

[21] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM Press/Addison-Wesley Publishing Co., 2000.

[22] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 159–170. ACM Press, 2001.

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

[24] Mark Segal and Kurt Akeley. OpenGL 2.0 specification. September 2004.

Paper III

Implementing an embedded GPU language by combining translation and generation

Calle Lejdfors and Lennart Ohlsson
Dept. 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.

Accepted for publication, SAC06

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.

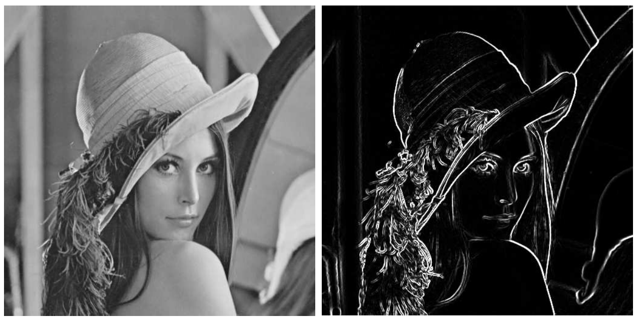


Figure 3.1: Sobel edge detected lena

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 3.1.

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, New York, NY, USA, 1998. ACM Press.

[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 Transactions on Programming Languages and Systems*, 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, New York, NY, USA, 2004. ACM Press.

[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, London, UK, 2000. Springer-Verlag.

[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`. 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, New York, NY, USA, 2004. ACM Press.

[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 IV

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

Calle Lejdfors and Lennart Ohlsson
Dept. 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.

Submitted for publication

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 [16] 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 [18], 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 [13]. Programming the GPU is done using specialized languages such as NVIDIA's Cg [15], Microsoft's HLSL [6], or GLSL by the OpenGL ARB [12].

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 [21]. 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 [14].

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)
```

Note that the position argument is omitted, the skin detector is applied to the whole image. The result is shown in Figure 4.1.

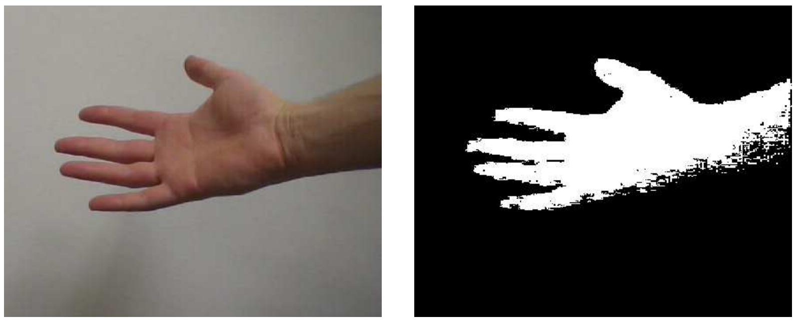


Figure 4.1: Skin detection

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 [14].

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 [20] 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))
```

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.

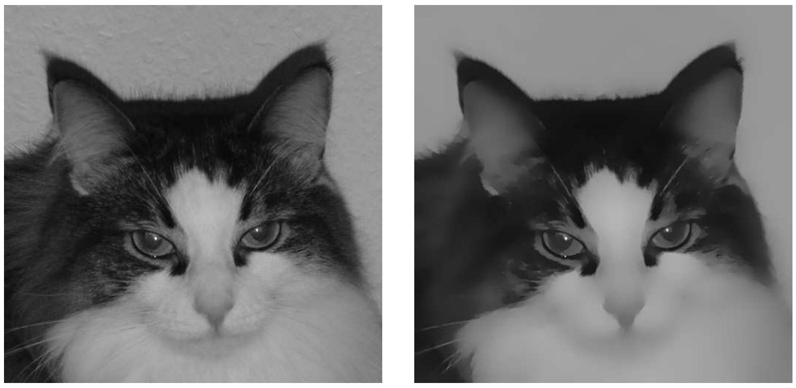


Figure 4.2: Perona-Malik anisotropic diffusion.

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 4.2 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)
```

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:

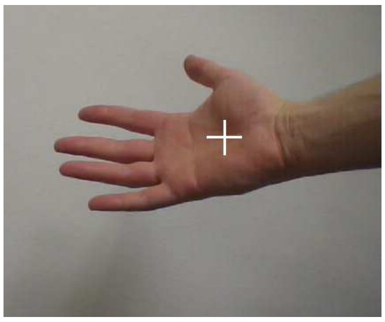


Figure 4.3: Center of mass

```
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 4.3.

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 [19]. 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:

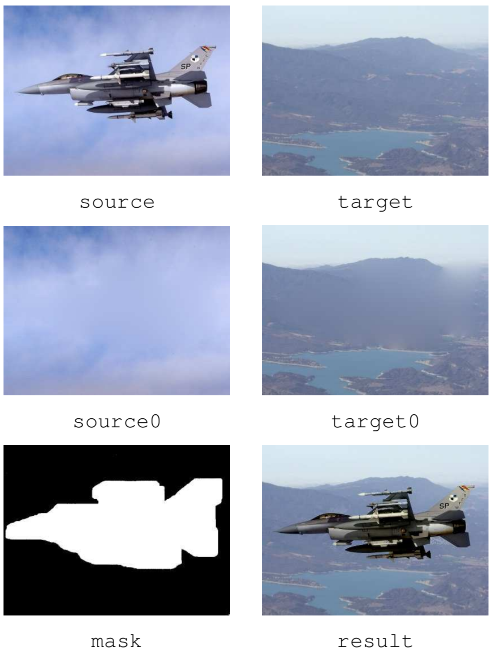


Figure 4.4: Seamless cloning

```
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 4.4.

The Laplace solver above will eventually reach a solution, but it converges very slowly. For the example in Figure 4.4 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 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):
    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]
```

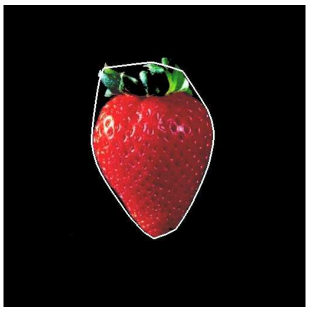


Figure 4.5: Contour detection using the snake algorithm

```
    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 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 4.5.

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 4.6. 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])
```

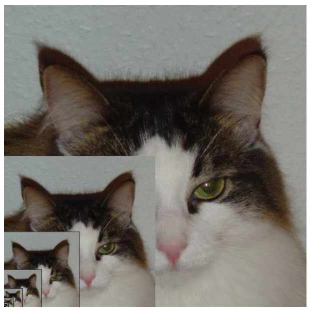


Figure 4.6: Block-wise reductions

```
    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.

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

	No. pixel ops.	No. texture accesses	Gpixel ops./s	Texture 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 diffusion	43	10	0.58	2.1
Laplace solver	18	4	0.60	2.8

Table 4.1: Performance figures for some of the examples

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 4.1 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 an 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.

This 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 [14].

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.* [17], 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., New York, NY, USA, 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, London, UK, 2000. Springer-Verlag.

[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(4es):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] John Kessenich, David Baldwin, and Randi Rost. The OpenGL shading language. `http://developer.3dlabs.com/documents/index.htm`. 3DLabs, Inc Ltd.

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

[14] Calle Lejdfors and Lennart Ohlsson. Implementing an embedded GPU language by combining translation and generation. *To appear in SAC'06 Programming Language track*, 2006.

[15] 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.

[16] Matlab. `http://www.mathworks.com/`.

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

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

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

[20] 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.

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