Functional Programming of Geometry Shaders

Jiří Havel

Faculty of Information Technology

Brno University of Technology

ihavel@fit.vutbr.cz

ABSTRACT

This paper focuses on graphical shader programming, which is essential for real-time rendering. Opposite to classical low level, structured languages, functional approach is used in this work and existing work is extended to cover geometry shader programming. The compiler is able to transform the program in a way that is hard to achieve with classical languages. The program is written for all pipeline stages at once and the compiler does the partitioning. This allows the programmer to focus on program semantics and let the compiler take care of the efficient execution. First, this paper describes shader stages as functions in a mathematical manner. The process of program partitioning and transformation to one of the classical languages is described. Several examples show the differences between functional description and equivalent structured code.

Keywords: Rendering, Shaders, Functional Programming

1 INTRODUCTION

Graphical hardware has changed greatly since first graphic accelerators. Its architecture evolved from fixed function pipeline, which became more and more configurable to today's fully programmable SIMT processors. However the programming is still low-level. The graphical processors lack complex control structures in exchange for raw computation power. The three most used languages for shader programming (GLSL [7], HLSL [8] and Cg [9]) mimic very closely the structure of the rendering pipeline.

The number of programmable stages of the rendering pipeline has risen from two to five in the latest accelerators. This means that the programmer must maintain even higher number of programs, executed at once on a single primitive, and ensure their compatibility. The interfaces between pipeline stages must be compatible not only in types, which the compiler can check, but also in passed values, which cannot be checked automatically. Packing the shader programs into one effect file solves this problem only partially. Effect files are only multiple shader programs, packed into one file with some additional information. When the effect file contains a code for multiple generations of graphical cards, the dependencies are even harder to maintain. This paper focuses on splitting one program to multiple parts and automatic generation of interfaces between them. Vertex, geometry and fragment shaders are the point of interest. The next two - hull and domain shaders were added for performance reasons only and might be addressed in future work.

Functional approach seems suitable for shader programming. Shaders transform data without any side effects and run massively parallel. Functional programs tend to be more abstract and allow the compiler to reorganize the code more than imperative languages. Because functional programs are referentially transparent, the order, in which the program is executed, does not matter. Every program transformation that preserves the output value is allowed. As shader programming favors speed over code clarity, this can help readability and maintainability without sacrificing performance. Significant parts of shader programs could be generated automatically.

Functional programming languages undergo a rapid development in recent years. Functional languages leave academic ground and slowly become well known like Microsoft's F#. Elements from functional languages like closures and lambdas are used in current mainstream languages (Python, C#). Ideas from functional programming like map-reduce [2] are used for programming parallel algorithms. These successes suggest that functional programming loses its reputation of being slow and is used for computationally intensive tasks. Because rendering is a computationally intensive task of different type, this paper explores the usability of functional programming for it.

Section 2 describes languages that were used as inspiration for this work. Section 3 shows shaders as functions from a mathematical point of view. Section 4 describes the transformation from functional program to C-like representation that is compatible with common shader languages. Section 5 summarizes the advantages of this approach and discusses open issues for the following work.

2 RELATED WORK

One of the interesting functional languages for shader programming is Vertigo [3], which was developed by Conal Elliott at Microsoft Research. Vertigo is an embedded language, focused on geometry and texture generation. Complex shapes are built from simple primitives and transformations by function composition. A significant part of the optimization is done by rewrite rules - common technique in functional programming, which is generally not applicable in imperative languages due to the lack of the referential transparency.

Another unfinished and interesting language for shader programming is Renaissance [1]. In this language, vertex and fragment shaders are specified as one program. The compiler splits the program and generates an interface between the vertex and fragment shader using simple rules that are based on expression frequencies and function linearity.

Expression frequencies correspond to pipeline stages, where the expression can be evaluated. Renaissance uses four frequencies - fragment, vertex, uniform and constant. Program is initially specified with fragment frequency and compiler determines lower frequencies for suitable expressions.

Function linearity is important for splitting vertex and fragment shader. For linear functions like addition is not important whether its input or output is interpolated over the rasterized primitive. This means, if input of linear function has vertex frequency, its output has vertex frequency too, so it can be safely moved to the vertex shader. Nonlinear functions like normalization can not be interpolated, so they must remain in the fragment shader. There exists another group of functions - partially linear - like multiplication. Its output can be interpolated if only one argument has vertex frequency and all other have frequency lower.

3 SHADERS AS FUNCTIONS

In this section and the following ones, a simplified Haskell [6] syntax will be used for program examples. Function types will be written in mathematical manner. For example $A \times B \rightarrow C$ means a function with a domains $A \times B$ (with two parameters of the type A and B) and a codomain C. Square brackets mean a list of values. It can be also an array, because the differences are not important here.

If we consider rendering as a function, the type of this function might be $U \times [A] \rightarrow [F]$. U denotes the uniform variables, textures and other rendering state, $[A]$ is the list of attributes of the rendered primitives and $[F]$ is the list of resulting fragments. This means the rendering takes the rendering state and the list of rendered vertices and transforms it to the list of fragments. These fragments are collected into the framebuffer. The rendering function can be split to three parts, equivalent to three pipeline stages.

The vertex shader does the transformation and lighting of all vertices. It has the type $U \times A \rightarrow V$. Because all vertices are processed identically, this function is simply mapped over input vertices. V is the vertex shader output.

The geometry shader follows the primitive assembly and takes one primitive consisting of one to six vertices. It has type $U \times [V] \rightarrow [[G]]$. It takes one primitive, which can be viewed as a list of vertices and outputs several triangle (or line) strips. Each triangle strip is simply a list of vertices, so the complete output is a list of strips. $[[G]]$ denotes the interface between geometry and fragment shader.

The primitives from the geometry shader are assembled, rasterized, values are interpolated over them and used as input for the fragment shader. The fragment shader has type $U \times G \rightarrow F$.

Aside from the mentioned parts or frequencies of computation (vertex, geometry, fragment), another two frequencies exist. It is the constant and uniform frequency. The expressions with constant frequency are evaluated at compile time. The expressions with uniform frequency transform uniform variables before rendering. For example HLSL preshaders have uniform frequency.

These frequencies not only assign expressions to pipeline stages. They also express relative cost of the computation and their cost increases from constant to fragment. Calculating expression at constant or uniform frequency is beneficial always. The limit is only the amount of constant and uniform registers.

The benefit of moving possible calculations from geometry to vertex shader is caused by Post Transform Cache. This cache is located after vertex shader and stores its outputs. In ideal case, each vertex has to be transformed only once, but in reality, the capacity of the cache is up to several tens of vertices. When drawing single triangles, the vertex shader is executed three times per triangle. When drawing triangle strips, VS is executed once per triangle (plus two times per strip). With indexed rendering of optimized meshes, VS can be executed less than once per triangle [10]. This means, we can safely move to vertex shader even calculations that could be performed on only one vertex of the triangle.

Moving calculation from fragment to geometry shader is beneficial in all cases, when interpolation is less costly, than calculation.

3.1 Expression Splitting

As was mentioned in section 2, the program can be split into stages automatically by the compiler. This simplifies the programmer's work as he does not need to maintain the interfaces between stages manually. Aside from simple splitting, some expressions can be automatically moved into parts with lower frequencies. The programmer can write calculations that logically belong together at one place and let the compiler move them apart to achieve more efficient execution.

This section describes the process of determining the frequencies of program expressions. In the beginning, only frequencies of shader inputs are known. Constants have constant frequency, uniform variables uniform frequency and vertex attributes have vertex frequency.

Selection of expressions with constant and uniform frequency is very similar. All function applications (function calls in structured languages) with constant frequency operands have constant frequency, too. Function applications with constant and uniform operands have uniform frequency. Listing 1 shows an example of vertex transformation and listing 2 equivalent code without declarations after frequency estimation and splitting.

```
uniform matrix4 model, view, projection
attribute vector3 position

-- original code
position' = projection*view*model*position
```

Listing 1: Original code of Uniform and Vertex shader

```
-- uniform part
tmp = projection*view*model

-- vertex part
position' = tmp*position
```

Listing 2: Uniform and Vertex shader after splitting

Vertex and geometry shader can be split at the point, where vertices of the input primitive are indexed. The function **at** is used for this purpose. Before indexing, the calculations are done for the complete stream of vertices. The function **at** can be moved automatically further into the geometry part. When all inputs of a function use the same index, this function can be evaluated in the vertex shader and its output can be passed into the geometry shader. All unary functions fulfill this criterion trivially.

Example in listing 3 calculates the distance of one vertex of each triangle from the camera (this can be used for example for LOD selection). Because **length** is an unary function, it can be moved into the vertex shader safely. Multiplication with one uniform argument acts as an unary function, too. The transformed program is shown in listng 4.

```
uniform matrix4 modelView;
attribute vector3 position;

-- original code
distance = length (modelView*(at position 1))
```

Listing 3: Original code of Vertex and Geometry shader

```
-- vertex part
tmp = length (modelView*position)

-- geometry part
distance = at tmp 1
```

Listing 4: Vertex and Geometry shader after splitting

When the geometry shader is not present, vertex and fragment shader can be partitioned fully automatically. This approach was used in Renaissance [1], but has some drawbacks. Because the program is practically written as fragment shader, it is hard to express calculations such as Gouraud shading. Also new versions of shaders provide multiple modes of value interpolation. Because of these reasons, I propose another method.

The point of splitting is specified by one of three functions - **smooth**, **linear** and **flat**. These names come from three interpolation modes on graphical cards. Functions **smooth** and **linear** can be moved further into fragment part by the same manner as in Renaissance. Calculations with all arguments with **flat** interpolation mode can be always moved into geometry (or vertex) shader, because no interpolation is performed. The centroid option does not complicate the transformation, so it is omitted here for simplicity.

The example in listing 5 shows a simplified calculation of specular lighting with phong shading. The geometry shader is omitted for simplicity. The transformation of the light vector is completely uniform. Multiplication is partially linear, so transformation of normal vector can be done in the vertex shader. Normalization is a nonlinear operation, so it must be left in the fragment shader. The light vector can be normalized in the uniform part, because it is not interpolated. The transformed code is shown in listing 6.

```
uniform matrix4 modelView, normalMatrix;
uniform vector3 lightVec;
attribute vector3 normal;

-- original code
norm = normalize (normalMatrix*(smooth normal))
lvec = normalize (modelView*lightVec)
color = norm 'dot' lvec
```

Listing 5: Original code of Vertex and Fragment shader

```
-- uniform part
lvec = normalize (modelView*lightVec)

-- vertex part
tmp = normalMatrix*normal

-- fragment part
norm = normalize (smooth tmp)
color = norm 'dot' tmpl
```

Listing 6: Vertex and Fragment shader after splitting

4 PROGRAM TRANSFORMATION

Automatic partitioning of the shader program is not the only important difference between conventional and functional approach. Very useful feature of functional languages are closures, partial application and higher order functions. Closures are nested functions with some variables defined inside the outer function. Partial application means that for example binary function can take one argument and can be used as unary function afterwards. Higher order functions are functions that take another function as a parameter or return it.

All these features significantly improve code expressiveness. Especially higher order functions offer the possibility of sharing code structure, that is hard to achieve or even not possible in structured languages. For complete implementation of these features, dynamic memory allocation is needed. Since the underlying hardware does not support it now, compiler must convert these features into equivalent structured code. The resulting code is often significantly less elegant, as will be shown in an example. The hardware also limits recursion, which must be limited to a form that can be automatically converted into loops. Sum-types, sometimes called discriminated unions, are also forbidden. Only product types - equivalent to C structures - are usable.

Enriched lambda calculus [5] can be used for program representation. This does not differ from other functional languages. The program is converted into a list of definitions which is topologically sorted. A definition is simply a named expression.

Because shaders do not have capabilities to support lazy evaluation, the program must be converted to an equivalent strict form. Both Vertigo and Renaissance solved this by complete substitution of all free variables in expressions. This approach is simple, but in the result, all common sub-expressions are lost.

In this paper a slightly more complicated approach is used. The program is lambda-lifted [4], so nested and anonymous functions are converted into C-like global functions. Substitution is done only to remove closures and partial applications, not for all variables. Lastly, all applications are merged into complete function calls.

Frequencies are estimated using rules from the previous section. For expressions without user-defined functions, the splitting is trivial. When a user-defined function is present, the frequencies inside it are estimated according to the parameter frequencies. Optionally, this function is also split into parts. Because of this splitting, library functions acting as one piece can be automatically split into multiple parts. This allows the use of library functions that silently cross the boundaries between shader stages and are both compact and effective.

Classical structured code can be now generated from the vertex and fragment part. The geometry part has one list of values for every output variable. To match the structure of the geometry shader, its output must be one list of structures containing every output variable. This conversion is in functional languages done by the function **zip**. This function takes multiple lists and converts it to a single list of structures. The length of the resulting list is the length of the shortest input list.

4.1 Larger example

This example illustrates the compilation of a more complex shader program. The uniform variables are *modelView* and *normalMatrix*. The vertex attributes are *vertex* and *normal*. The required output variables are *position* with frequency geometry and *color* with frequency fragment. The source code without declaration of variables is shown in listing 7. This program transforms the input vertices and normals, splits the triangles into four parts as shown in figure 1 and calculates simple diffuse lighting. The splitting is described by function **gen**. This function is used for position, normal and light vector identically. A real program would add some modification after, but for this example, simple subdivision will suffice; any such complication would not affect the compilation process.

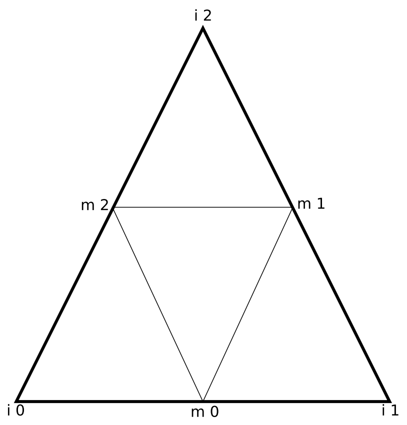

Figure 1: Subdivision of a triangle in the geometry shader in listing 7. The input vertices *i* and generated vertices *m* correspond to the list in the function **gen**.

```
tr_pos = modelView*vertex
tr_norm = normalMatrix*normal

gen i = [[i 0, m 2, m 0, m 1, i 1], [i 2, m 1, m 2]]
        where m x = (i x + i (x+1)%3)/2

position = gen (at ftransform)
lvec = lightPos - (smooth (gen (at tr_pos)))
norm = smooth (gen (at tr_norm))
color = (normalize lvec) 'dot' (normalize norm)
```

Listing 7: Code for triangle transformation, subdivision and simple shading

The definitions are already sorted, so no reordering is needed. All expressions depend only on previous definitions. Lambda lifting splits the function **gen** and creates a new function **gen_m**. These two functions are now C-like global functions. The resulting code is shown in listing 8.

```
tr_pos = modelView*vertex
tr_norm = normalMatrix*normal

gen_m i x = (i x + i (x+1)%3)/2

gen i = let m x = gen_m i x in [[i 0, m 2, m 0, m
      1, i 1], [i 2, m 1, m 2]]

position = gen (at ftransform)
lvec = lightPos - (smooth (gen (at tr_pos)))
norm = smooth (gen (at tr_norm))
color = (normalize lvec) 'dot' (normalize norm)
```

Listing 8: Shader after lambda-lifting. Only the function **gen** differs from listing 7.

Partial applications of functions like *m* in the function **gen** or usages of the function **at** are substituted to places where the remaining arguments are applied. By this substitution, specialized lists for variables *position*, *lvec*, and *norm* are created. The function **gen** itself and the lifted function **gen_m** are removed as a dead code. The resulting code is shown in listing 9.

```
tr_pos = modelView*vertex
tr_norm = normalMatrix*normal

position =  [[at ftransform 0, ((at ftransform 2) +
     (at ftransform (2+1)%3))/2 ...
lvec = lightPos - (smooth [[at tr_pos 0, ...
norm = smooth [[at tr_norm 0, ...
color = (normalize lvec) 'dot' (normalize norm)
```

Listing 9: Shader without partial applications and closures

Expression frequencies are estimated, expressions are split, constant expressions are evaluated and common subexpression elimination is done. Vertex and fragment parts are prepared for code generation. Geometry part needs zipping together, which is trivial. Listing 10 shows this situation.

```
-- vertex frequency
tr_pos = modelView*position
tr_norm = normalMatrix*normal
tmp1 = ftransform
tmp2 = lightPos - tr_pos

-- geometry frequency
position = [[at tmp1 0, ((at tmp1 2) + (at tmp1
     0))/2 ...
lvec = [[at tmp2 0, ((at tmp2 2) ...
norm = [[at tr_norm 0, ...

-- fragment frequency
color = (normalize lvec) 'dot' (normalize norm)
```

Listing 10: Code parts for each stage of the rendering pipeline

Listing 11 shows the generated code. The interface between the vertex and geometry shader are the variables *tr_norm*, *tmp1* and *tmp2*. The interface between the geometry and fragment shader are the variables *lvec* and *norm*.

```
//vertex shader
tr_pos = modelView*position;
tr_norm = normalMatrix*normal;
tmp1 = ftransform;
tmp2 = lightPos - tr_pos;

//geometry shader
position = tmp1[0];
lvec = tmp2[0];
norm = tr_norm[0];
emitVertex();
position = (tmp1[2] + tmp1[0])/2;
lvec = (tp2[2] + tmp2[0])/2;
//... too long

//fragment shader
color = dot(normalize(lvec), normalize(norm));
```

Listing 11: Code equivalent to listing 7 in the target structured language

The final code does not contain the interfaces between shader stages, because they are straightforward. The code for the geometry shader was shortened, because all vertices are generated nearly identically. In classical languages, the structure of generated vertices cannot be shared, so the resulting code must be written by hand or generated by some preprocessing tool.

5 CONCLUSION AND FUTURE WORK

This paper presented a functional approach to the geometry shader programming. This approach has some interesting properties that are hard to achieve in conventional structured languages.

One program is written for all shader stages and the compiler does the necessary partitioning and interface generation. This simplifiers the programmer's work, as he can write the code, where it logically belongs and let the compiler move it for an efficient execution.

Higher order functions allow the programmer to write the code more abstract. Abstract code often tends to be shorter and more readable. The code sharing is possible at a level that is hard to achieve by traditional languages.

Automatic partitioning of program also helps modularity. Library functions can be viewed as single blocks by the programmer, but parts of them can be executed in different stages of the pipeline.

These properties significantly improve the shader programming. However it is not likely that so massive shift of used paradigm could occur. Because of that, following work will focus on selecting useful parts that could be used to extend existing languages.

REFERENCES

[1] Chad Austin and Dirk Reiners. Renaissance : A functional shading language. *Graphics Hardware*, 2005.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, December 2004.

[3] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[4] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. pages 190–203. Springer-Verlag, 1985.

[5] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[6] Simon Peyton Jones. Haskell 98 language and libraries: The revised report, 2003.

[7] Khrohos Group. *OpenGL API and Shading Language Specification*, August 2009.

[8] Microsoft Corporation. *DirectX Reference*, 2009.

[9] NVIDIA Corporation. *NVIDIA GPU Programming Guide*, May 2009.

[10] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), August 2007.