Rendering Pipeline Modelled by Category Theory

Jiří Havel
Faculty of Information Technology
Brno University of Technology
ihavel@fit.vutbr.cz

Adam Herout
Faculty of Information Technology
Brno University of Technology
herout@fit.vutbr.cz

ABSTRACT

This paper describes basic concepts from category theory, which are commonly used in functional programming. These concepts are applied to shader programming and to the rendering pipeline and the whole rendering pipeline is formally modelled using category theory. This model can be used for more abstract and formal approach to shader programming. Mathematical formalization of the rendering pipeline and its stages can be helpful in shader compiler design, for proving algorithms, complexity analysis, and other tasks.

Keywords: Rendering, Shaders, Category Theory

1 INTRODUCTION

Category theory [9, 3] is an abstraction of mathematical structures and relations between them. It started as a "generic abstract nonsense", but now it is heavily used not only in mathematics, but also in computer science and especially functional programming. Many categorical concepts give rise to common pieces of code used to combine computations together. >From these abstractions, especially functors and monads are almost ubiquitous [7].

Although the rendering pipeline is programmable, its overall structure is basically fixed. The design of programming languages for shader programming perfectly corresponds to the structure. Some experimental shading languages tried to offer a slightly more abstract way. Two notable examples are the Gpipe[1] library for Haskell and Sh [5, 4] for C++. Both are, however, limited to OpenGL 2 or DirectX 8 functionality.

The Sh library views shaders as objects, that can be combined using two operations – serial and parallel composition. These two operations, however, perfectly correspond to a category with products, which will be described in the next section.

The GPipe library achives the same functionality by a different approach. It operates on streams of primitives and transforms them by series of functions – using functors from the category theory.

The rest of the paper will discuss correspondences between those approaches. Section 2 will introduce a category-based model of the rendering pipeline stages and extend programming of pipeline stages to a description of the whole pipeline. Section 3 will raise the abstraction of the pipeline description to a composition of stream transformers. Section 4 summarizes the introduced categories as a model of the rendering pipeline.

[1] http://www.haskell.org/haskellwiki/GPipe

2 CATEGORY WITH STREAMS

A category (**C**) consists of a set of objects (obj **C**) and a set of arrows or morphisms (arr **C**) that link these objects. An arrow $f : A \to B$ has a domain A and a codomain B from obj **C**. The set of arrows from A to B is denoted as $\mathrm{Hom}_{\mathbf{C}}(A,B)$. Arrows must be composable (1a), every object must have one identity arrow (1b) (1c) and arrow composition must be associative (1d).

$$\forall f : A \to B, g : B \to C \; \exists (g \circ f) : A \to C \tag{1a}$$

$$\forall O \in \mathrm{obj}\ \mathbf{C} \; \exists 1_O : O \to O \tag{1b}$$

$$\forall f : A \to B, 1_B \circ f = f \circ 1_A = f \tag{1c}$$

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{1d}$$

A classical example is the category **Set**, whose objects are sets and arrows are mappings between those sets.

N-ary functions and tuples are modelled using products – composite objects. Product P is an object consisting of objects $O_i, i = 0, \ldots N$ with projection arrows $(p_i : P \to O_i)$ that extract its member objects. For every object Z that has arrows $a_i : Z \to O_i$ to all members of P, exactly one arrow a_P to the product exists, such that $a_i = p_i \circ a_P$, see Figure 1. An exaple of the product type can be the structure data type, that is a product of its elements. When describing function, the arrow a_P is a combination of functions returning the elements of a structure to a function returing the whole structure.

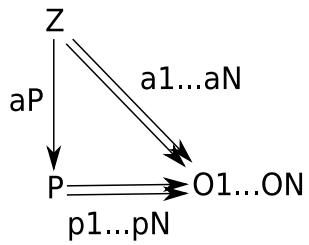

Figure 1: Product in a category

Let us define a category of GPU programs called **Gpu**. Objects of **Gpu** are basic and structured

datatypes of graphical shaders and arrows are functions (both built-in and composite). Both structured datatypes (structure and array) fullfil the requirement for product types[2] as do the basic vector types. With streams from the following subsection, every possible shader will be an arrow in **Gpu**.

2.1 Streams are Functors

Functor is a structure-preserving mapping between categories. Functor $F : \mathbf{A} \to \mathbf{B}$ between categories $\mathbf{A}$ and $\mathbf{B}$ consists of a mapping $F_{\text{obj}} : \text{obj}\,\mathbf{A} \to \text{obj}\,\mathbf{B}$ and mappingss $F_{A_1,A_2} : \text{Hom}_{\mathbf{A}}(A_1,A_2) \to \text{Hom}_{\mathbf{B}}(F_{\text{obj}}(A_1),F_{\text{obj}}(A_2))$ for every pair of objects A_1, A_2 from $\mathbf{A}$. Functors must also preserve arrow composition and identity arrows.

Homogenous abstract data types like lists, stacks, or queues are constructed by functors. The F_{obj} creates structured types from the basic ones. The mappings for arrows convert simple arrows to arrows working with new types – usually doing the same for every element of the new type. In functional programming, this family of mappings is denoted by *map*. $map : (X \to Y) \to (F(X) \to F(Y))$, where F is a functor, i.e., for a given arrow on the simple data type, *map* defines the corresponding arrow on the structured data type.

In shader programming, one abstract data type is ubiquitous – the stream of values (vertices, primitives, fragments). It is a sequence of elements, that is, however, never handled directly, but implicitly by the streaming nature of the rendering pipeline. Stream types in the category **Gpu** are constructed using a functor $Stream : \mathbf{Gpu} \to \mathbf{Gpu}$. This functor creates streams of elements of any basic type (and recursively streams of streams). For every X and Y from obj **Gpu**, the mapping for arrows is simply defined as

$$map : (X \to Y) \to (Stream(X) \to Stream(Y))$$
$$map(f)(x_1,\ldots,x_n) = (f(x_1),\ldots,f(x_n)).$$

The mapping $map(f)$ transforms every stream element by the function f, see Figure 2.

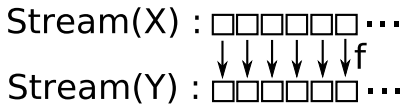

Figure 2: Functorial transformation principle.

Streams are not the only functors in the category **Gpu**, but arrays and basic vector types are functors as well. For these data types, *map* works similarly to its stream counterpart.

For categories $\mathbf{A}$ and $\mathbf{B}$ a category $\mathbf{B}^{\mathbf{A}}$ exists: the *functor category*, whose objects are functors from $\mathbf{A}$ to $\mathbf{B}$. Arrows of this category are called *natural transformations*. For every functor F and G from $\mathbf{B}^{\mathbf{A}}$, natural transformation $\phi : F \to G$ and arrow $f \in A$, must $\phi \circ F(f) = G(f) \circ \phi$.

Natural transformations change only the structure of an abstract data type, but the elements of the type are left unchanged. For example, a transformation between an array of streams and a stream of arrays is a natural transformation. Natural transformations are heavily used in the following subsections.

2.2 Streams are Monads

Monad is a special type of functor used in functional programming to represent computations and control structures, to embed side effects, or model a processing pipeline.

Monads in category theory is a functor F, together with two natural transformations $\eta : 1_{\mathbf{C}} \to F$ and $\mu : F^2 \to F$ ($F^2 = F \circ F$). The corresponding triplet in functional programming consists of a functor F and mappings $unit : A \to F(A)$ and $join : F(F(A)) \to F(A)$ [6].

Mapping *unit* creates the monad type from one element and *join* merges two layers of the monad to one. In the category **Gpu**, the transformation $unit : X \to Stream(X)$ creates a stream with a single element. The transformation $join : Stream(Stream(X)) \to Stream(X)$ joins a stream of streams to one single stream by concatenation.

In functional programming, the *bind* transformation is used more than *join* and it better describes the properties of monads.

$$bind : (X \to F(Y)) \to (F(X) \to F(Y))$$
$$bind(f) = join \circ map(f)$$

In **Gpu**, the transformation $bind : (X \to Stream(Y)) \to (Stream(X) \to Stream(Y))$ uses the provided mapping to transform a stream. The type shows that every element of the input stream is used to create a new stream and such new streams are concatenated together. In other words, $0 - N$ elements can be created from every single input element and the input elements are processed separately, as shown by Figure 3.

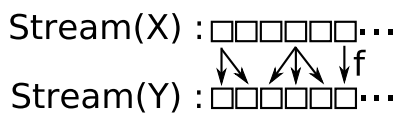

Figure 3: Monadic transformation schema.

A perfect example of monadic processing of a stream is the geometry shader. From every primitive of the input stream, zero or more primitives are generated and those new streams are concatenated. Also, the fragment shader can be described by a monad, as it can output empty streams or streams with a single element.

[2] Objects like sums (discriminated unions) or exponentials (partially applied functions) are hard to express on GPU, so will not be used in this paper.

2.3 Streams are Comonads

Functor and monad are sufficient to describe stream transformers that access single elements of a stream. For accessing multiple elements, a categorical dual to a monad can be used – the comonad. Comonads can represent some information in a context. In the case of streams, the context of each element are the neighboring elements.

As dual functor to a monad, comonad is a functor with two natural transformations in opposite direction as the monad. These are $extract : F \to 1_{\mathbf{C}}$ and $duplicate : F \to F^2$. The functional programming forms are $extract : F(X) \to X$ and $duplicate : F(X) \to F(F(X))$. Mapping *extract* discards the context of a value and *duplicate* duplicates the context for every input.

Similarly to monad function *bind*, comonad has its dual *extend*.

$$extend : (F(X) \to Y) \to (F(X) \to F(Y))$$
$$extend(f) = map(f) \circ duplicate$$

As the type suggests, *extend* can not change the element count, but contrary to *bind*, the output elements depend on the context of the input elements as shown by Figure 4.

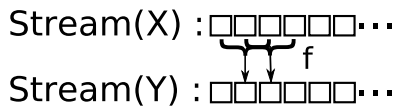

Figure 4: Comonadic transformation schema.

In **Gpu** several implementations of the comonad for streams are possible. Preceding or following elements can form a context of a stream element – the underlying implementation can be a delay link for example. The actual implementation is not important for the scope of this paper.

The primitive assembly can be viewed as a comonad (followed by a monad); however, because of the independence on actual stream contents, primitive assembly can be also modelled as a natural transformation. The tesselation control or hull shaders have also the comonadic structure, although they do not operate on stream but on array with index.

3 PIPELINE CATEGORY

The previously introduced category **Gpu** mixes the pipeline structure and the stages implementation. We can construct a category, that models only the pipeline structure and abstracts the actual implementation of the stages.

For a category **C** with monad M, which contains arrows of type $A \to M(B), A, B \in \text{obj}\,\mathbf{C}$, exists another category **K**,

$$\text{obj}\,\mathbf{K} = \text{obj}\,\mathbf{C}$$
$$\text{Hom}_{\mathbf{K}}(A,B) = \text{Hom}_{\mathbf{C}}(A, M(B))$$

The arrow composition in **K** is defined as $g_{\mathbf{K}} \circ f_{\mathbf{K}} = bind(g_{\mathbf{C}}) \circ f_{\mathbf{C}}$, so kategory **K** expresses composition of stream transformations. This category is called *Kleisli category* of **C**.

Dual to Kleisli category, also the *CoKleisli category* **L** for every category **C** exists, with a comonad N.

$$\text{obj}\,\mathbf{L} = \text{obj}\,\mathbf{C}$$
$$\text{Hom}_{\mathbf{L}}(A,B) = \text{Hom}_{\mathbf{L}}(N(A), B)$$

Similarly, the arrow composition in **L** is defined as $g_{\mathbf{L}} \circ f_{\mathbf{L}} = g_{\mathbf{C}} \circ extend(f_{\mathbf{C}})$.

For the category **Gpu**, we can construct a *pipeline* or *stream category* **Pipe**. Objects of this category are basic and structured shader types and arrows are functions of type $Stream(X) \to Stream(Y)$. Arrows fall to three groups.

- $map(f)$, f is an arrow of **Gpu** without streams.
- $bind(g)$, g is an arrow of the Kleisli category of **Gpu**.
- $extend(h)$, h is an arrow of the CoKleisli category of **Gpu**.

This category provides a superset of all stream transformations possible on GPU. In functional programming, this kind of structure is called *Arrows* [2, 8].

From the axioms for functor, monad and comonad [1], the following equivalences can be derived:

$$map(f) \circ map(g) = map(f \circ g) \quad (2a)$$
$$map(f) \circ bind(g) = bind(map(f) \circ g) \quad (2b)$$
$$bind(g) \circ map(f) = bind(g \circ f) \quad (2c)$$
$$map(f) \circ extend(h) = extend(f \circ h) \quad (2d)$$
$$extend(h) \circ map(f) = extend(g \circ map(f)) \quad (2e)$$
$$extend(h) \circ bind(g) = map(h) \circ duplicate \circ join \circ map(g) \quad (2f)$$
$$bind(g) \circ extend(h) = bind(g \circ h) \circ duplicate(x) \quad (2g)$$

When applied to the shader programming, these equations are intutive. Equation (2a) shows that two vertex-shader-like stages can be composed to one. Equations (2b) and (2c) show the composition of a geometry-shader-like stage with the following or preceding vertex shader. Equations (2d) and (2e) show the same for a comonadic shader.

The category **Gpu** has product types – tuples (structures) of basic types or streams. For arrows of type $map(f)$, the tuples of streams are isomorphic to streams of tuples. Therefore, also streams of tuples have the properties of product types. The language *Sh* uses these properties for parallel composition of shaders.

When arrows of type $bind(f)$ are considered, the product properties are lost. As outputs of two arrows can be differently structured, they cannot be generally merged together without mutual affection. This limits *Sh*-style parallel composition capabilities to vertex shader, tesselation evaluation, and fragment shader without discard.

The stream category can contain only streams of tuples. Because tuples in **Pipe** are not products, the arrows generally can not be combined in parallel. This parallel composition is limited to *functorial* and *comonadic* arrows.

4 MODEL OF THE RENDERING PIPELINE

Randering pipeline can be described using two categories. **Gpu** models complete GPU functionality from implementation of shader stages to the whole pipeline structure. Pipeline category **Pipe** models the pipeline structure by composition of simple shaders without considering their implementation.

The simple shaders can be classified according to their capabilities. The pipeline stages can be classified similarly.

- *Functorial* shaders change only single stream elements. The results do not depend on the context and the stage cannot change the stream's structure. Existing stages: Vertex shader, Tesselation evaluation shader, Fragment shader without discard.
- *Monadic* shaders can expand and remove elements. Their input is limited to one element. Existing stages : Rasterization, Fragment shader with discard, Fragment tests, Geometry shader, Hardware tesselator.
- *Comonadic* shaders can process the context of the element – the input consists of several elements (possibly the whole stream). The output is, however, limited to one element. Existing stages : Primitive assembly.

Using equivalences from (2), the stages can be composed from multiple simple shaders. Functorial stages can be composed only from functorial shaders. The (co)monadic stages can be composed from both (co)monadic and functorial shaders, as every (co)monad is a functor.

Also *functorial* and *comonadic* stages can be constructed using parallel composition. Both *Sh* and *GPipe* use only equivalence (2a). Following equivalences can be used to extend the model capability to cover geometry and tesselation shaders.

5 CONCLUSION

This paper introduced a model of the rendering pipeline using category theory. Although the mathematics in this paper is not novel, it is not commonly seen in the field of computer graphics. Two categories are defined and used: **Gpu** describing pipeline capabilities, structure and implementation and **Pipe** for abstacting the pipeline structure and composition from simple shaders.

The formalism introduced in this paper can be used for classification of different shader operations and for automatic optimization of shader programs on inter-stage level. Notably the equivalences from equation 2 form rewrite rules for moving computations between different stages of the rendering pipeline. However application of these rules is not trivial. Following work will focus on searching suitable rules, probably using genetical algorithms.

The model is inspired by two actual shader languages and will be used for their extension. The **Pipe** category can be also possibly used to describe generic stream processing.

REFERENCES

[1] Neil Ghani, Christoph Lüth, Federico De Marchi, and John Power. Algebras, coalgebras, monads and comonads, 2001.

[2] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[3] Saunders Mac Lane. *Categories for the Woring Mathematician*. Springer, 1971.

[4] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM.

[5] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[6] Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23. IEEE Computer Society Press, 1988.

[7] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[8] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[9] R. F. C. Walters. *Categories and Computer Science*. Cambridge University Press, 1992.