Shading Languages

Ari Silvennoinen

Apri 12, 2004

Introduction

The recent trend in graphics hardware has been to replace fixed functionality in vertex and fragment processing with programmability [1], [2], [3]. This allows developers to create unforeseen visual complexity that used to possible with only offline renderers. Vertex processing involves the operations that occur at each vertex, including transformation and lighting. Fragment processing consists of the operations that occur after rasterization, most notably the sampling of textures and assigning the sampled values to each fragment.

Evolution of programmable shading

Assembly languages for graphics hardware programming have been around for awhile and they share the same set of inadequasities as assembly languages for traditional computer hardware, namely:

- lack of portability
- error-prone syntax
- tedious to program

The next natural step in the realm of real-time rendering was to come up with higher level languages.

The limitations of a fixed shading model were first realized by Robert Cook, who proposed a flexible tree-like shading model in 1984 [4]. Each shader is organized in a tree-like structure, where the innder nodes are operations and child nodes are inputs to the operations. The final color is produced by evaluating the tree.

Cook's work later influenced the design of the RenderMan Shading Language, the most established shading language for offline rendering.

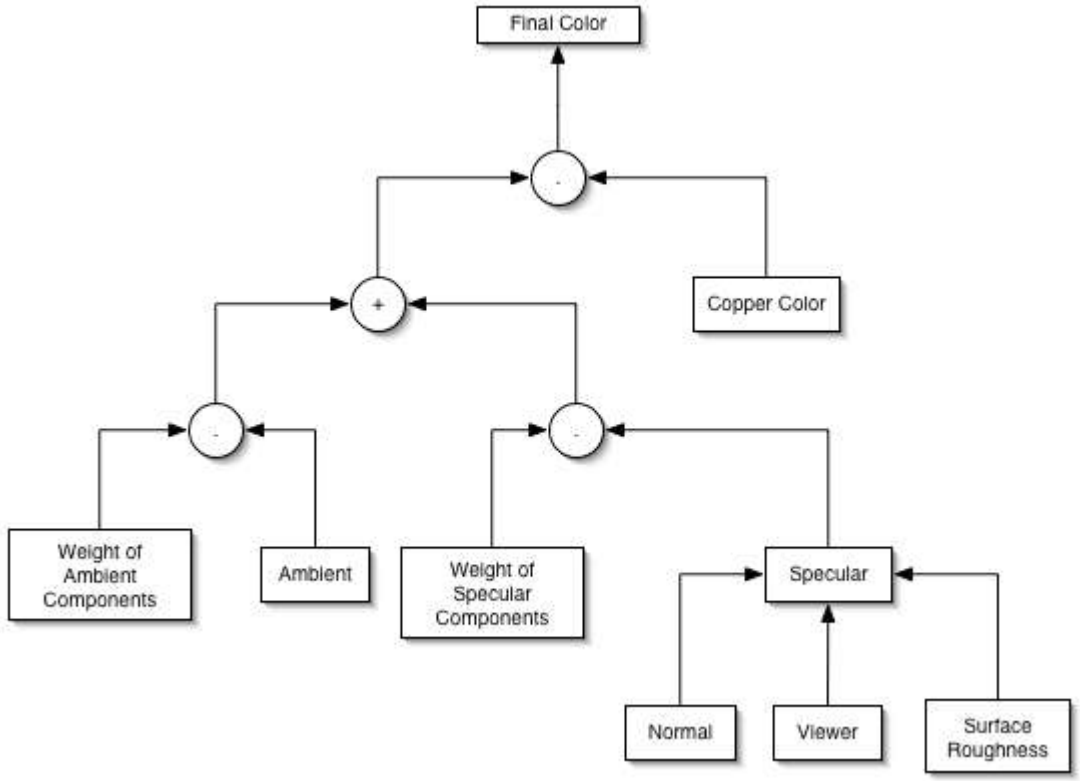

Illustration 1A Shade Tree Example, from Rob Cook's original SIGGRAPH paper

Renderman uses curved surfaces in contrast to triangles as it's geometric primitives. These surfaces are tesselated to “micropolygons” that are smaller than a pixel. The programmable shader operates on the vertices of these polygons. This approach is not feasible on current graphics hardware, due to the lack of support for automatic tesselation of curved surfaces to micropolygons. Traditionally graphics hardware performance has

not been high enough to allow for the use of pixel-sized polygons.

The first programmable graphics hardware architechture, PixelFlow, was developed by Marc Olano et al at UNC in the mid-1990s [5]. However, the project failed commercially due to high cost of the described architechture.

Consequent advances in graphics hardware guided the evolution of real-time programmable shading from hardware vendor-specific assembly languages towards high level shading languages, most notably the OpenGL Shading Language and Microsoft's High Level Shading Language.

OpenGL Shading Language

The OpenGL Shading Language has been developed by the OpenGL Architechture Review Board (ARB), an independent consortium which governs the OpenGL specification. The language was designed with the following goals in mind [3]:

- hardware independence
- high performance
- ease of use
- long lifespan
- ease of implementation

The language itself is based on the C-programming language. Shading programs written in the OpenGL Shading Language share a common structure with the C-equivalents. The entry point of a set of shaders is the function *void main ()*.

Data Types And Data Abstraction

The OpenGL Shading Language supports vector types for floating-point, integer and boolean values, as well as scalars. In addition floating-point matrix types are also included. Matrix types allow a convenient representation of linear affine transformations, commonly used in 3D graphics. Basic operators such as addition,

substraction, multiplication and inverse are defined for both vectors and matrices.

Another set of domain specific basic data types are called Samplers. Samplers are a special type used to access texture maps. The OpenGL Shading Language has support for 1D, 2D, 3D, shadow and cube map textures.

Information flow between the application and shaders differs considerably from common programming environments. Information is transferred to and from a shader by accessing built-in variables and user defined variables. User defined variables can be associated with one of the following qualifiers:

- *attribute* For frequently changing information, from application to a vertex shade
- *uniform* For infrequently changing information, for vertex and fragment shaders
- *varying* For interpolated information passed from vertex to fragment shader
- *const* For compile time constants

```
uniform float Kd;
attribute vec4 Dir;
vec4 a(1, 0, 0, 1);
vec4 b(0, 0, 1, 1);
vec4 c = a*b;
float d = dot (a.xyz, b.yzw);
uniform sampler2D baseTexture;
vec4 color = texture2D (baseTexture)*vec4 (Kd,
Kd, Kd, 1);
```

Table 1Usage of basic data types

Variables behave similarly to C++ in terms of declarations and scoping. Type *void* is a special type used to declare a function with no return value. The OpenGL Shading Language disallows implicit type promotion. It provides operators for excplicit type

conversion.

Unqualified variables may be initialized when declared, that is attribute, uniform and varying variables cannot be initialized when declared. The OpenGL Shading Language provides constructors fro all the built-in types, excluding samplers, in addition to structures.

The OpenGL Shading Language has built-in support for arrays and structures of any type.

Control Flow And Control Abstraction

The entry points to shaders are functions named *main*. A program can consist of a vertex shader, a fragment shader or from both a vertex shader and a fragment shader. In the latter case, the shaders will have two functions named *main*; one for the vertex shader and another one for the fragment shader.

The OpenGL Shading Language has support for similar control flow mechanisms as C++. Iteration is supported by *for*, *while* and *do-while* looping constructs, which are semantically and syntactically like their C++-counterparts. Looping constructs allow arbitrary exit points with *break* and *continue* statements. They behave semantically and syntactically as in C/C++.

Selection is done with *if* and *if-else* constructs, which operate on boolean expressions. Boolean expressions can be chained with logical *and* (&&), *or* (||) or *xor* (^^) operators. These operations follow the short-circuit evaluation semantics like C/C++. The OpenGL Shading Language supports also the ternary :?-operator, which behaves exactly as in C++.

The OpenGL Shading Language has a special statement, *discard,* that prevents the frame buffer from being updated by the current fragment.

Functions can be declared, defined and used much in the same way as in C++. Either a function definition or a declaration must be in the scope before it can be called. Parameter types are

statically checked. Functions can be overloaded by parameter type, excluding differentation based solely on the return type. Exiting from a function is done with the return statement. Functions declared as nonvoid must return a value, whose type must exactly match the declaration. Recursion is not supported, neither directly nor indirectly.

Functions follow the call by value-return calling convention. Parameters can be declared as in, out or inout, and the different semantics are described below:

- *in* Copy in, don't copy back out; writable within the function
- *out* Copy only out; readable but undefined at entry to function
- *inout* Copy in and copy out
- *const* Function cannot write to it

```
vec4 ComputeDiffuse (in vec3 normal, in vec3
light, in float3 lightcolor, in float Kd, in
vec3 ambientColor)
{
    vec3 color = max (0, dot (normal, light))
*lightcolor*Kd + ambientColor;
    return vec4 (color, 1);
}
```

Table 2A Function example

The OpenGL Shading Language defines an extensive set of built-in functions, which are/will be hardware accelerated. Built-in functions include support for:

- trigonometric functions
- exponential functions
- geometric functions
- matrix and vector functions
- texture access and fragment processing functions

- noise functions

References

[1] David S. Ebert et al, *Texturing&Modeling: A Procedural Approach*, Morgan Kaufman 2003.

[2] Tomas Akenine-Möller, Eric Haines, *Real-Time Rendering 2ed,* A K Peters 2002

[3] Randi J. Rost, *OpenGL Shading Language*, Addison Wesley 2004

[4] Robert L. Cook, *Shade Trees*, ACM SIGGRAPH Computer Graphics, Volume 18, Issue 3, July 1984

[5] Lastra, Anselmo, Steve Molnar, Marc Olano, and Yulan Wang, *Real-Time Programmable Shading*, Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, CA, April 9-12, 1995), ACM SIGGRAPH, New York, 1995.