Programmable Graphics Pipeline Architectures

Author: Martin Ecker
Project Website: http://xengine.sourceforge.net
Last Modified: 24. March 2003 20.15

1.1 Introduction

Current consumer graphics hardware, like NVIDIA's GeForce 3 and GeForce 4 chipset family or the ATI Radeon 8500 series, offers the possibility of replacing the fixed-function rendering pipeline with user-developed programs, usually referred to as *shaders* or *shader programs*. Newer generation cards, like the NVIDIA GeForce FX, ATI Radeon 9700, or cards based on 3Dlabs' P10 chip, provide an extended programmability model that offers a larger instruction set and even dynamic flow control, transforming GPUs into highly programmable processors. Future generations of graphics hardware can be expected to further increase programmability. In the not too distant future, GPUs will become general purpose processors that cannot only perform graphics-oriented operations, but also other computations such as encryption in parallel to the host CPU.

In this paper and also in XEngine the term *shader* is used instead of *program* or *shader program* to refer to the pieces of code that program certain parts of the pipeline. There are mainly historical reasons for this. RenderMan [Hanr90] and DirectX 8.0 use the term shader. In addition, it is the most common term found today. Some say that the term shader has the connotations of only representing colour operations and has nothing to do with vertices, which certainly is a valid argument. However, neither RenderMan nor DirectX make this distinction and also the OpenGL 2.0 drafts chose the term shader over program. Various OpenGL extensions for low-level shading languages, such as ARB_vertex_program and ARB_fragment_program, use the term program, however. This should be kept in mind when reading the corresponding specifications.

Two computational frequencies are supported in current graphics hardware, per vertex and per fragment. As such, there are two different kinds of shaders, vertex shaders and fragment shaders. Vertex shaders get executed for each vertex that passes through the

rendering pipeline and can change the vertex position or any other user-defined vertex attributes specified per-vertex, such as normals, colours or texture coordinates. Fragment shaders get executed for each fragment and have access to the texture sampling stages in the pipeline. Additionally, a fragment shader receives the computational results of a vertex shader as inputs. These inputs must include the vertex position in clip coordinates and can include any other user-defined attributes, such as generated or modified texture coordinates or colours. Using the inputs from the vertex shader, fragment shaders read texels (= texture samples) from the texture stages and combine them in some way with the other inputs to form the final colour value that gets passed on to the final stages of the rendering pipeline, like stencil and depth testing. Most new fragment shading languages also allow modification of the depth value that gets used in the depth test.

The majority of shaders are nowadays written in low-level, rendering API-dependent assembly languages that are internally compiled by the rendering API or graphics driver into machine code of the GPU. In the case of vertex shaders, if the GPU doesn't support all requested features, they might also get compiled into specialized CPU machine code, for example, utilizing Intel SSE or AMD 3DNow! technology, to emulate the shader on the CPU. The capabilities of these shader assembly languages are very limited, and the available opcodes are not very flexible or general purpose. Few registers are available, and the number of commands per shader is limited to only a few dozens.

However, in the foreseeable future, a massive capacity increase of the programmable features of graphics hardware can be expected, and the use of high-level shading languages instead of cryptic assembly mnemonics will become standard. A first step in this direction has already been taken by Stanford University with its Stanford Real-Time Shading Project [Mark01][Prou01], sponsored by various vendors like NVIDIA and SGI. The Stanford project designed a C-like shading language on top of OpenGL and developed a compiler for consumer graphics hardware, such as the GeForce 3. Using the experience from the Stanford project, NVIDIA, together with Microsoft, developed its own C-like shading language called Cg [Kirk02] with a compiler capable of compiling to various OpenGL low-level shading languages and to the DirectX 8.0 and 9.0 assembly shading languages. Cg is now also a part of Direct3D 9.0, where the language is called HLSL. The OpenGL 2.0 white papers [Rost02][Bald03] also outline a C-like, high-level shading language called Glslang that will be one of the main improvements of OpenGL 2.0, expected for release in mid-2003.

1.2 Architecture Overview

Using programmable shaders is a good way of taming the complexity of current rendering APIs. The ray-tracing industry has long discovered the advantages of shading languages and has already used languages like the RenderMan Shading Language [Hanr90] for a couple of years. However, for current hardware the RenderMan Shading Language is too complex. It is nevertheless a viable goal for future generation hardware to be able to use it in real-time visualization.

In [Olan98] Olano proposed an abstract, real-time graphics pipeline decomposed into a number of user-programmable stages and implemented a procedural C-like shading lan-

guage for it. This was one of the first attempts at developing a real-time shading architecture with an accompanying shading language. The pipeline stages used for Olano's system were model, transform, primitive, interpolate, shade, atmosphere, and warp. In his thesis Olano also provided a concrete implementation of his ideas for PixelFlow [Eyle97], an expensive, highly programmable graphics system for generating high-speed, highly realistic images. Most of the pipeline stages proposed by Olano are also found in current fixed-function pipeline hardware or can be mapped to them. Figure 1.1 shows an abstracted block diagram of the typical pipeline stages of current fixed-function hardware.

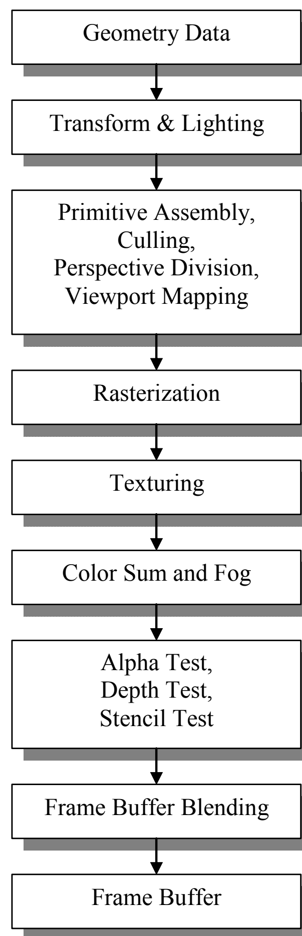

Figure 1.1: Abstract Block Diagram of Current Fixed-Function Pipelines

As opposed to the many stages in Olano's abstract graphics pipeline, current programmable graphics hardware only offers two types of programmable pipeline stages that combine most of the stages proposed by Olano. Having to deal with only two programmable stages reduces the complexity of GPUs and also has the advantage for developers that they are only faced with two programming models. The two stages are the vertex processing stage and the fragment processing stage, programmed by so-called vertex shaders and fragment shaders, respectively. Vertex shaders operate at the vertex level and replace the transform and lighting part of the pipeline. Fragment shaders operate at the

fragment level and replace parts of the fragment processing pipeline. The following sections describe each of the two shader types in detail.

1.2.1 Vertex Shaders

Vertex shaders get executed for each vertex that passes through the pipeline. A vertex shader is a program that has exactly one vertex as input and one vertex as output. A vertex in this context is a structure composed of a number of vertex attributes, one of which must be the vertex position. Other vertex attributes include normal vector, primary and secondary colour, texture coordinates, or any other user-defined value that is required for the per-vertex computations in the vertex shader. Vertex shaders cannot remove vertices from a primitive, nor can they add new vertices. Furthermore, they can never operate on several vertices (or primitives) at the same time.

When a vertex shader is used the following parts of the vertex processing fixed-function pipeline are not active, and changing a render state that affects these parts will have no effect on the vertex shader:

- Transformation from world space to clipping space
- Normalization
- Lighting and materials
- Texture coordinate generation
- In some shader execution environments user-defined clipping planes are also disabled

All other parts of the fixed-function pipeline are not replaced, in particular:

- Primitive assembly
- Frustum culling
- Perspective division
- Viewport mapping
- Backface culling

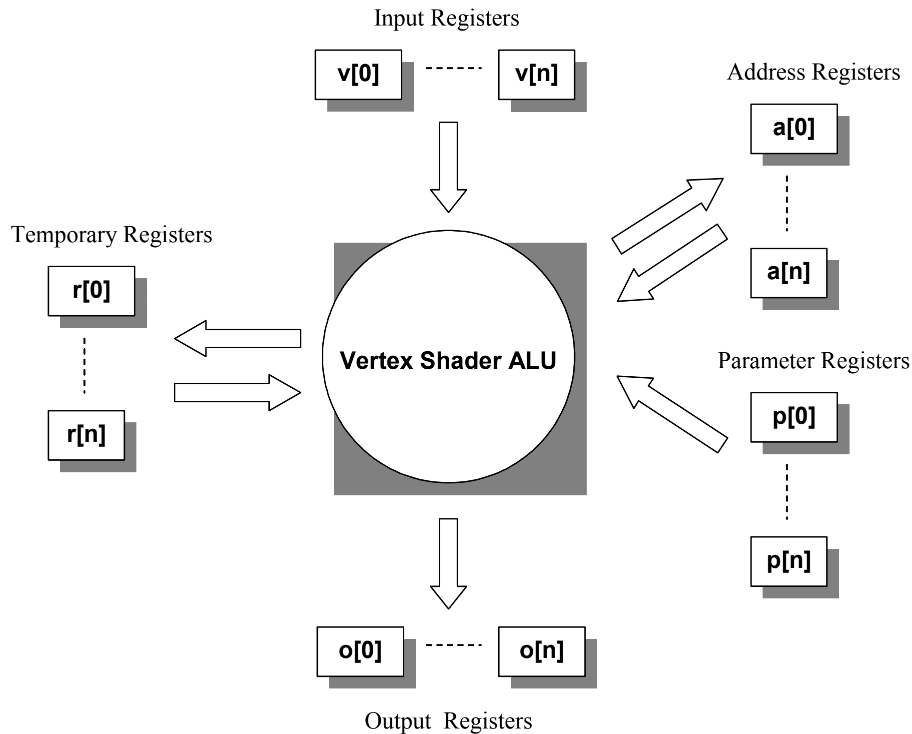

Figure 1.2: Vertex Shader Execution Environment

Even though most shading languages, especially low-level, assembly-like languages, define their own shader execution environment, the general architecture always closely resembles the architecture shown in figure 1.2. The vertex shader has access to a number of register files, some of which are read-only or write-only. In current hardware designs these registers are usually four-component vectors of floating-point values, but this is not a necessity. Newer generation hardware also supports integer and boolean registers [Micr02].

The shader can read the vertex attributes from a relatively small number of read-only input registers. Using a typically large number of read-only parameter registers and a small number of temporary registers the shader then performs its computations. The parameter registers contain values that do not change per vertex, but only change once every frame or once every couple of frames. Examples of values that are usually stored in the parameter registers are the combined world-view-projection matrix (or any variation of it), light directions, light positions, or matrix palettes used for indexed vertex blending. A small number of address registers can also be used by the shader to perform indexed relative addressing into the array of parameter registers. These registers can generally not be directly read in the shader, but only used for relative addressing.

Finally, the shader writes its results to a number of write-only output registers. These output registers have a pre-defined semantic meaning, such as the transformed, homogeneous vertex position, texture coordinates, and vertex colours. These results are then

passed on to the next stages of the fixed-function pipeline, and might eventually be used by a possibly activated fragment shader at a later stage in the pipeline.

1.2.2 Fragment Shaders

Fragment shaders get executed per fragment during the rasterization phase in the graphics pipeline. A fragment is a point in window coordinates produced by the rasterizer with associated attributes, such as interpolated colour values, a depth value, and possibly one or more sets of texture coordinates. A fragment modifies the pixel in the frame buffer at the same window space location based on a number of parameters and conditions defined by the pipeline stages following the rasterizer, such as the depth test, the stencil test, or a fragment shader. Sometimes the notion of fragment is mistaken for the notion of pixel. However, a pixel is only the final colour value written to the frame buffer, and each pixel in the frame buffer usually corresponds to multiple fragments. Some of these fragments get discarded because of e.g. the depth test; others might get combined to form the final pixel colour.

Logically, fragment shaders operate on fragments just before they reach the final stages of the rendering pipeline, such as the alpha, depth, and stencil tests. The fragment shader receives the vertex shader outputs interpolated across a primitive as input and delivers a single colour value and a depth value that gets passed on to the final stages of the pipeline as output.

When a fragment shader is used, the following parts of the fixed-function fragment pipeline are not active:

- Texture access
- Texture application and blending
- Fog and colour sum (some execution environments still allow fixed-function fog computations to follow after the fragment shader)

However, the following functionality is not subsumed by fragment shaders:

- Shading model
- Alpha test
- Depth test
- Stencil test
- Frame buffer blending
- Dithering

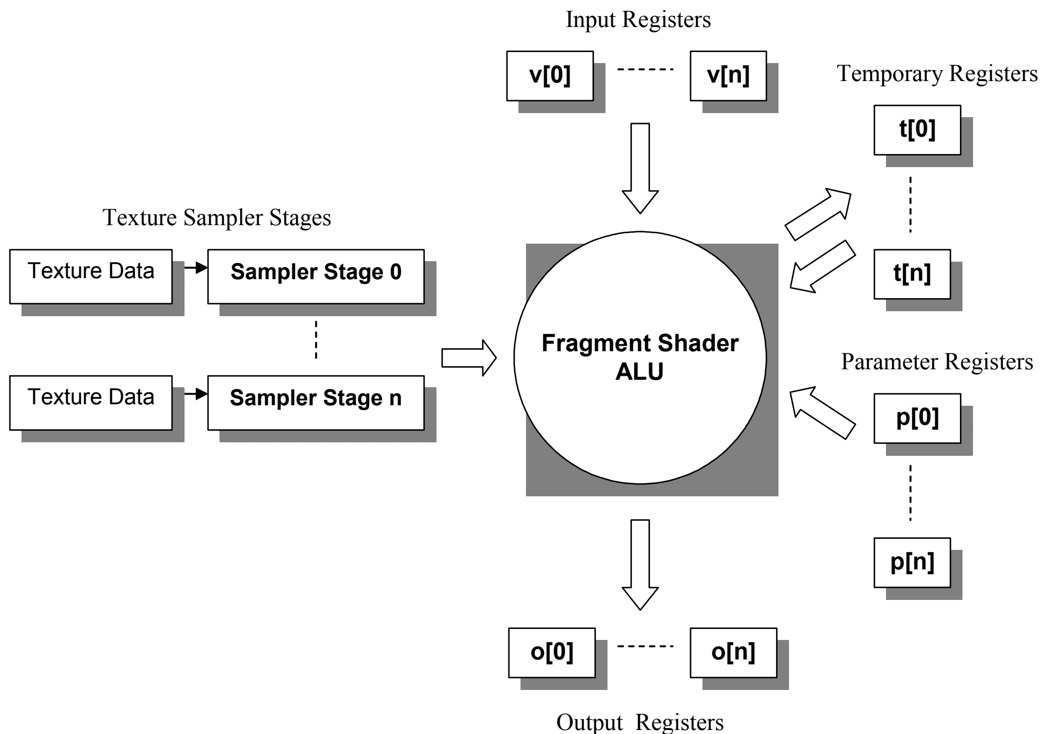

Figure 1.3: Fragment Shader Execution Environment

Similarly to vertex shaders, the fragment shader execution environment is slightly different depending on the shading language used. However, the basic architecture remains the same and usually closely resembles figure 1.3. Just like vertex shaders, fragment shaders have access to a number of register files, where some registers are read-only and some are write-only. The input registers contain the interpolated vertex shader results, such as the fragment's colour values or texture coordinates, and are read-only for the shader. Additionally, the fragment shader can look up filtered texture values using texture sampler stages. The fragment shader can either use the interpolated texture coordinates passed in in one of the input registers or texture coordinates computed directly in the shader to sample the texture. Dependent texture reads are also possible, allowing more advanced effects, such as per-pixel lighting. Using the input register values and sampled texture values the shader then computes its results and stores them in the write-only output registers. These output registers have a pre-determined semantic meaning. The typically supported outputs are the final fragment colour that will be used as pixel colour in the frame buffer, if the fragment passes the alpha, depth, and stencil tests, and a possible fragment depth value that will be used in the depth test for the fragment. Note that currently no fragment shading language or fragment shader execution environment supports address registers, which is mostly due to the fact that current execution environments do not offer a large enough number of parameter registers to justify address registers.

1.3 Low-Level Shading Languages

This section provides an overview of currently available low-level, assembly-like shading languages. Low-level shading languages resemble common assembly languages for general purpose CPUs, with the difference that their instruction set is very limited and contains special instructions that only make sense for graphics programming. The base data type for registers in all the shading languages presented subsequently is a four component vector of floating-point values. All the languages offer SIMD-like instructions that can work on all four components of a register simultaneously. Typical instructions of this category are component-wise addition, component-wise multiplication, or the four-component dot product which, among other things, can be used to compute the result of matrix-vector multiplications, which is typically one of the most common operations in graphics programming.

While most high-level shading languages provide the same syntax for writing vertex and fragment shaders, this is usually not the case for low-level languages. In fact, there is no assembly-like shading language available at the moment that can be used to write vertex shaders and fragment shaders. Instead the languages are always specific to one type of shader, where the language for fragment shaders is usually less powerful than the language for vertex shaders. However, looking at newer low-level shading languages, the same instructions are becoming available in both vertex and fragment shaders. It can be expected that future languages will use almost the same instruction set and syntax.

The following subsections discuss the various currently available low-level shading languages. The first ever shading language for consumer graphics hardware, the Direct3D 8 shading language, is introduced. Then the OpenGL equivalents, NV_vertex_program and ARB_vertex_program, are discussed. Finally, more advanced languages for newer generation hardware are presented, such as the new versions of the low-level shading languages of Direct3D 9.0, NV_vertex_program2, and ARB_fragment_program. Since the Direct3D 8.0 shading languages were the first languages to appear, the discussion of them will introduce a lot of concepts and ideas that are also valid for the other shading languages.

1.3.1 Direct3D 8 Shading Languages

Version 8.0 of DirectX, Microsoft's multimedia API for Windows, was the first major 3D graphics API to introduce a programmable pipeline and a vertex and fragment shader assembly-like shading language to go with it [Micr01]. The vertex shader language of Direct3D 8 can, by design, replace the entire transform and lighting pipeline stage. The fragment shader language can replace the multitexturing and blending pipeline stage of previous versions of Direct3D. It slightly extends the texture blending capabilities, but does not offer much computational power and has a rather rigid syntax and a large number of restrictions.

1.3.1.1 Direct3D 8 Vertex Shader Assembler

The execution environment for vertex shaders closely follows the general execution environment shown in figure 1.2. All registers are four-component floating-point registers. There is only a single address register, which is also a four-component floating-point register, but only the x component can be used for indexed relative addressing into the array of parameter registers. Before the value of the address register is used to perform relative addressing, it is rounded down to the next integer number. In DirectX, the parameter registers are also called constant registers. Table 1.1 summarizes the available registers for DirectX 8 vertex shaders. In the table, if a register name contains the letter *n* in italics, it represents a group of registers where the *n* is replaced by an index from 0 to *count* minus one, where *count* is the number of available registers as noted in the corresponding column of the table.

Table 1.1: Registers in the DirectX 8 Vertex Shader Execution Environment

Name	Type	Usage	Count
a0	address register	write/use	1
v*n*	input registers	read-only	16
r*n*	temporary registers	read/write	12
c*n*	parameter registers	read-only	96
oPos	homogeneous position output register	write-only	1
oD0	primary colour output register	write-only	1
oD1	secondary colour output register	write-only	1
oPts	point size output register	write-only	1
oFog	fog colour output register	write-only	1
oT*n*	texture coordinate output registers	write-only	8

Twelve output registers for passing the computed results on to the next stages of the pipeline are available. As listed in table 1.1, the names of these output registers all begin with the lower-case letter *o*. The output registers have fixed names and pre-defined semantics. Every vertex shader must at least write the vertex position in homogeneous clip coordinates to the *oPos* output register. The homogeneous vertex position can be computed by storing the combined world-view-projection matrix in four parameter registers, and then multiplying the vertex position, which the shader receives in one of the input registers in local coordinates, with that matrix. The two vertex colour registers, *oD0* and *oD1*, and the eight texture coordinate registers are interpolated across a primitive during rasterization and passed on to a possible fragment shader or to the fixed-function multitexturing stage of the pipeline.

The general syntax for a vertex shader instruction in DirectX 8 (and all of the subsequently discussed assembly-like shading languages) looks like this:

```
opcode destReg , srcReg1 [, srcReg2] [, srcReg3]
```

where `opcode` represents the instruction opcode, such as `mov`, `add`, `mul`, or `dp3`, `destReg` represents the name of the destination register for the instruction, and `srcReg1`, `srcReg2`, and `srcReg3` are the names of the source registers for the instruction. The square

brackets indicate that the last two source registers are only used with certain instructions. There are two types of instructions, general instructions that use up exactly one instruction slot, and macro instructions, like the `m4x4` vector-matrix multiplication instruction, that get expanded to a number of general instructions and therefore require more than one slot. A vertex shader can use a maximum of 128 instruction slots. Table 1.2 lists the available general instructions in the DirectX 8 vertex shading language. Newer vertex shading languages typically support the same instructions and possibly some new instructions, such as `sin` or `cos` for directly computing the sine and cosine of a value.

Table 1.2: Direct3D 8 Vertex Shading Language General Instructions

Opcode	Arity	Description	Example
add	Binary	Adds the two sources and stores the result in the destination register.	`add r0, c0, v0`
dp3	Binary	Calculates the three-component dot product of the two source vectors and replicates the result to all four components of the destination register.	`dp3 oPos, r0, r1`
dp4	Binary	Calculates the four-component dot product of the two source vectors and replicates the result to all four components of the destination register.	`dp4 r1, r0, c0`
dst	Binary	Calculates the distance vector between the two source vectors.	`dst r0, v2, c5`
expp	Unary	Computes the exponential function with base two with low-precision.	`expp r0, c0.x`
lit	Unary	A special instruction that calculates lighting coefficients that can be used in per-vertex lighting computations.	`lit r3, r0`
logp	Unary	Computes the logarithmic function with base two with low-precision.	`logp r1, c2.y`
mad	Ternary	Multiplies the first two sources with each other and then adds the third source.	`mad r0, c0, c1, v0`
max	Binary	Computes the component-wise maximum of the two source vectors.	`max r2, r0, r1`
min	Binary	Computes the component-wise minimum of the two source vectors.	`min, r3, c0, c1`
mov	Unary	Moves the contents of the source register into the destination register.	`mov oD0, v1`
mul	Binary	Multiplies the two sources in a component-wise manner.	`mul r0, c0, v0`
rcp	Unary	Computes the reciprocal of the source scalar.	`rcp r1, v0.x`
rsq	Unary	Computes the reciprocal square root of the source scalar.	`rsq r0, c0.y`
sge	Binary	Sets the destination to 1.0 if the first source operand is greater than or equal to the second source operand, or to 0.0 otherwise.	`sge r0, v0, c0`
slt	Binary	Sets the destination to 1.0 if the first source operand is lower than the second source operand, or to 0.0 otherwise.	`slt r0, v0, c2`
sub	Binary	Subtracts the two sources from one another.	`sub oPos, r0, v3`

In addition to the general instructions the DirectX 8 vertex shading language has a number of macro instructions that get expanded to general instructions. For example, the `m4x4` macro instruction gets expanded to four `dp4` general instructions. Table 1.3 lists the available macro instructions.

Table 1.3: Direct3D 8 Vertex Shading Language Macro Instructions

Opcode	Arity	Description	Example
exp	Unary	Computes the exponential function with base two with high-precision.	`exp r0, c0.z`
frc	Unary	Computes the component-wise fractional portion of the source vector.	`frc oD1, c1`
log	Unary	Computes the logarithmic function with base two with high-precision.	`log r2, v0.w`
m3x2	Binary	Computes the product of the source vector and the 3x2 matrix specified by the second source register, which must be a constant register.	`m3x2 r0, v0, c0`
m3x3	Binary	Computes the product of the source vector and a 3x3 matrix specified by the second source register, which must be a constant register.	`m3x3 r0, v0, c5`
m3x4	Binary	Computes the product of the source vector and a 3x4 matrix specified by the second source register, which must be a constant register.	`m3x4 r0, v0, c0`
m4x3	Binary	Computes the product of the source vector and a 4x3 matrix specified by the second source register, which must be a constant register.	`m4x3 r5, v0, c0`
m4x4	Binary	Computes the product of the source vector and a 4x4 matrix specified by the second source register, which must be a constant register.	`m4x4 r0, v0, c3`

Additionally, the language supports modifiers that can be used on source and destination registers at no additional runtime cost. The negation modifier, indicated by putting a minus sign in front of a register name, allows negating a source register before it is read. The source swizzle mask can be used to swap or replicate the components of a source register in any way. For example, `r0.wzyx` changes the regular order of the components of the vector register r0 to the exact opposite order. Similarly, `r0.x` replicates the x component into all four components. Note that source swizzle masks do not actually change the contents of the source register, but only use the components in the way specified by the swizzle mask when the register is read. Finally, the destination register mask can be used to mask out writing to certain components of the destination register of an instruction. For example, using `r0.xz` as destination register will only write the result of the instruction to the x and z components.

Skimming through the instruction set, it becomes obvious that a couple of important and rather useful instructions are missing, for example a division instruction or instructions to compute the sine and cosine of a value. However, with a bit of trickery these instructions can be emulated. Division can be performed by using the rcp and mul instructions:

```
; scalar division r0.x = r1.x / r2.x
rcp r0.x, r2.x; compute 1 / r2.x
mul r0.x, r1.x, r0.x
```

The sine, cosine, and other functions can be approximated by using the corresponding Taylor series [Wlok01][Lind00a]. As will become evident in the following sections, newer generation shading languages have these instructions already built in as general instructions and do not require such tricks.

As was mentioned before, the transform and lighting functionality of the fixed-function graphics pipeline can be completely replaced by a vertex shader [Lind00b]. For example, the following vertex shader emulates fixed-function pipeline functionality for one enabled directional light using one set of texture coordinates. As with the fixed-function pipeline, the main program must pass in vertices that have a position and a vertex normal as vertex attributes. The vertex position in local object coordinates is contained in register v0, the vertex normal in v1, and the texture coordinates in v2. The main program must also provide the combined world-view-projection matrix in the parameter registers c0 to c3, the inverse transpose of the world matrix in registers c4 to c7, the light direction vector in world coordinates in register c8, a diffuse material colour in c9, a global ambient colour in c10, and the constant value 0 in c11.x. Whenever any of these values changes, the application must reset them in the corresponding parameter registers of the vertex shader. The shader performs the lighting calculations in world space.

```
; transform the vertex from local object space to clip space
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

; transform the normal from local to world coordinates
dp4 r1.x, v1, c4
dp4 r1.y, v1, c5
dp4 r1.z, v1, c6
dp4 r1.w, v1, c7

; normalize the normal vector
dp3 r1.w, v1, v1
rsq r1.w, r1.w
mul r1, r1, r1.w

; normalize the light direction vector
mov r2.xyz, c8
dp3 r2.w, c8, c8
rsq r2.w, r2.w
mul r2, r2, r2.w

; perform the lighting computation
; color = ambient + diffuse * max(0, dot(normal, light direction))
dp3 r3.x, r1, r2
max r3.x, r3.x, c11.x
mad oD0, c9, r3.x, c10

; simply pass through the texture coordinates
mov oT0, v2
```

In section 1.4.2, we shall later examine what this shader looks like in the high-level shading language Cg to see the benefit of using a high-level shading language.

1.3.1.2 Direct3D 8 Pixel[1] Shader Assembler

The fragment shading language of Direct3D 8 is a rather primitive and restricted language that replaces the multitexture stage of the fixed-function pipeline. There are five different versions of the language. Versions 1.0 to 1.3 are based on the same execution environment. Higher versions up to 1.3 successively add instructions and lift some restrictions of earlier versions. Version 1.4, which was introduced with DirectX 8.1, uses a different execution environment and represents a break with previous versions of the language. It was introduced for ATI's new consumer graphics card at that time, the Radeon 8500.

There are two main types of instructions in the Direct3D 8 pixel shader assembly language: texture addressing instructions and arithmetic instructions. The two types cannot be mixed, and texture addressing instructions must be specified before any arithmetic instructions in the shader. This holds true for all versions of the pixel shader language in Direct3D 8. Newer fragment shading languages, such as ARB_fragment_program, which will be discussed later in section 1.3.6, do not impose such restrictions and texture addressing and arithmetic instructions can be used anywhere in a fragment shader.

Texture addressing instructions use so-called texture registers to sample textures. They replace the texture fetching functionality of the fixed-function pipeline. When a texture addressing instruction is executed, the texture coordinate set indicated by the number of the specified texture register is used to sample a texture. The texture sample is then stored in the texture register and can be used by other instructions of the shader. Some texture addressing instructions perform various transformations on the input texture coordinates and use the computed coordinates to sample the texture. Also dependent texture reads are possible, using the result of a texture lookup to lookup another texture.

Note that, just like the texture coordinate set, the sampler stage to be used is also indicated by the number of the specified texture register for language versions up to 1.3. Version 1.4 lifts this restriction and uses the number of the destination register to determine the sampler stage to be used. Therefore, with pixel shader language versions prior to 1.4, it is not possible to use the same set of texture coordinates with multiple texture sampler stages. So there is a one-to-one relationship between the texture coordinate set and the texture sampler stage. Texture coordinate set 0 cannot be used with sampler stage 1 but only with sampler stage 0.

Arithmetic instructions are used to combine the interpolated vertex colours that are passed in as input parameters to the fragment shader and the texture samples obtained via the texture addressing instructions. Thus arithmetic instructions replace the texture combining functionality of the fixed-function pipeline. The fragment colour and a possible depth value used for the subsequent depth test represent the final outputs of the fragment shader. The available arithmetic instructions in the Direct3D 8 pixel shader assembly language are `add`, `sub`, `dp3`, `dp4`, `mul`, `mov`, `mad`, and a couple of other fragment shader-

---

1. Direct3D does not differentiate between the notions *fragment* and *pixel*. Therefore fragment shaders are called "pixel shaders" even though Direct3D's pixel shaders actually perform fragment shading as described in section 1.2.2.

specific instructions. The instructions are used just as the corresponding vertex shader instructions presented in table 1.2.

Unlike newer fragment shading languages, the pixel shader language versions 1.0 to 1.3 of Direct3D 8 offer a large variety of texture addressing instructions that not only perform texture fetching but also various arithmetic computations. There are, for example, instructions to perform a matrix transformation on a set of texture coordinates before using it to sample a texture. This design decision was necessary because it is not possible to arbitrarily mix texture addressing instructions with arithmetic instructions. However, it later proved to be a bad design choice, since these calculations could also be performed by regular arithmetic instructions, and adding new computations for texture coordinates would require new texture addressing instructions, which would lead to an even larger and more complex instruction set. Therefore, in newer fragment shading languages, also in pixel shader language version 1.4 of Direct3D 8, there are only a small number of texture instructions that exclusively fetch texture samples and do not perform any computations on texture coordinates. Transforming texture coordinates is done by simply using arithmetic instructions on the texture coordinates before using them to sample a texture.

Even though the computational power of the Direct3D 8 fragment shading language is very limited, it is already capable of computing some interesting per-pixel lighting effects, such as per-pixel bump mapping using Blinn's formula [Kilg00]. To evaluate Blinn's formula, only addition, three-component dot product, multiplication, and division operations are required. The division is required to normalize vectors used in the lighting calculations. However, since a division instruction is not available to Direct3D 8 fragment shaders, tricks have to be used to achieve the desired results. To normalize vectors a so-called normalization cube map texture can be used which contains unit-length vectors encoded as RGB triples. The not normalized vector is now interpreted as 3D texture coordinate to sample the cube map texture. The result of this texture lookup is the normalized vector. Newer fragment shading languages have a division instruction, or at least a reciprocal function, so that vector normalization can easily be performed in a fragment shader.

1.3.2 NV_vertex_program

NV_vertex_program [Kilg02a] is an OpenGL extension that defines a vertex shader execution environment with an accompanying low-level shading language. The NV prefix in its name indicates that the extension was developed by the graphics hardware vendor NVIDIA. At the time of this publication being issued, the NV_vertex_program extension is available on all NVIDIA graphics cards of the GeForce series, the Matrox Parhelia graphics card, newer 3Dlabs cards with the P10 GPU, and Mesa, the OpenGL look-a-like software renderer, versions 4.1 and up. In an OpenGL-typical manner the extension refers to what is called *vertex shader* in this paper as *vertex program*.

The execution environment of NV_vertex_program is basically the same as the environment of DirectX 8 vertex shaders and not computationally more powerful. It can be seen as the OpenGL equivalent to Direct3D 8 vertex shaders. Except for a couple of new instructions that can, however, be emulated by using other instructions in Direct3D, and the omission of macro instructions, the instruction sets of shading languages are the same

as presented in table 1.2. The same register modifiers, such as destination register masks, source register negation, and source register swizzle masks, are supported. Also the number of available input, output, temporary, address, and parameter registers is the same as for the Direct3D 8 vertex shader execution environment. Syntactically, the mnemonics used in NV_vertex_program are upper-case as opposed to lower-case, all instructions have to be ended by a semicolon, and the input and output register names are specified by using array indexing syntax, such as `o[HPOS]` instead of `oPos` or `v[2]` instead of `v2`.

The three additional instructions that NV_vertex_program offers over the Direct3D 8 vertex shading language are listed in table 1.4.

Table 1.4: New Instructions in NV_vertex_program

Opcode	Arity	Description	Example
ABS	Unary	Assigns the component-wise absolute value of the source vector to the destination register.	`ABS o[HPOS], c[1]`
DPH	Binary	Calculates the four-component dot product of the two source vectors assuming, however, that the fourth component of the first source vector is 1.0. The result is replicated to all four components of the destination register.	`DPH R1, R0, c[0]`
RCC	Unary	Calculates the reciprocal value of the source scalar and clamps the result to the range $[2^{-64}, 2^{64}]$, if the reciprocal value is positive, or $[-2^{64}, -2^{-64}]$ otherwise. The reason for this clamping is to keep a certain amount of floating-point precision for subsequent calculations.	`RCC R0.x, R0.x`

NV_vertex_program also introduces so-called position invariant vertex shaders. A vertex shader is called *position invariant* when it produces the exact same clip coordinate position for a vertex as would the conventional, fixed-function pipeline. This is important for multi-pass rendering techniques where some passes use the fixed-function pipeline. In OpenGL, no precision requirements for the vertex transformations from local object space to clip coordinates are specified. Therefore it is easily possible that in a multi-pass algorithm which uses both vertex shaders and the fixed-function pipeline the same vertex position in local object coordinates might end up having different clip coordinates. For these cases an option that guarantees position invariance has been introduced. A position invariant shader is not allowed to write a vertex position in clip coordinates to the corresponding output register. Instead, the vertex position is computed implicitly by the GPU using the same computation as used by the fixed-function pipeline.

1.3.3 ARB_vertex_program

The ARB_vertex_program OpenGL extension [Brow02a] was officially approved by the ARB, the Architectural Review Board, in June 2002. It is the culmination of previous vendor efforts, most notably NVIDIA's NV_vertex_program, to specify a vertex shading language for OpenGL. ARB_vertex_program closely resembles NV_vertex_program in many ways. The instruction set and instruction syntax are almost the same. Position

invariant shaders are supported. Also the execution environment is closely related. ARB_vertex_program offers the same kind of input, output, temporary and address register, but differentiates between two kinds of parameter registers, so-called program local parameters and program environment parameters. The former are parameters local to a vertex shader whose values are lost once the shader is no longer the current vertex shader. The latter retain their values and can also be read by other vertex shaders. Furthermore the number of available registers is not limited per se anymore. Instead, an application can query how many parameter or address registers are available.

Unlike the vertex shading languages presented so far, ARB_vertex_program does not have fixed register names anymore. Instead, register names must be declared using special declaration statements before they are used. For example:

```
ATTRIB pos = vertex.position;
OUTPUT outpos = result.position;
TEMP myTempReg;
ADDRESS myAddressReg;
PARAM mvp[4] = { state.matrix.mvp };
```

The above piece of code declares a vertex attribute register called `pos` that contains the vertex position, an output register called `outpos` that maps to the homogeneous position computed by the shader, a temporary register called `myTempReg`, an address register called `myAddressReg`, and a special parameter register called `mvp` that uses an automatic state tracking, a new feature introduced with ARB_vertex_program. Note that some register names are already declared implicitly and need not be re-declared in a vertex shader. This includes all the input and output registers which all carry the prefix `vertex` and `result`, respectively, in their names and the local and environment parameter registers that are called `program.local[i]` and `program.env[i]` with *i* being a number between zero and the maximum number of available parameter registers.

Automatic state tracking allows a vertex shader to automatically track various state variables of the OpenGL state machine. One such variable is `state.matrix.mvp`, as used in the above example, which gives the shader access to the combined model-view-projection matrix as set by the application using the standard OpenGL calls like `glLoadMatrix`. In the other vertex shading languages discussed so far, state tracking was a tedious task for the application and required setting certain parameter registers manually whenever a state needed by the vertex shader had changed. In addition to the various standard OpenGL matrices, such as the model-view, the projection, or the texture matrices, other useful state that can be tracked includes all light parameters set via `glLight`, the texture coordinate generation planes set via `glTexGen`, and the material parameters set via `glMaterial`.

As mentioned above, the instruction set of ARB_vertex_program is almost the same as NV_vertex_program, and therefore also almost the same as the DirectX 8 vertex shading language. It lacks NV_vertex_program's `RCC` reciprocal clamp instruction, but adds five new instructions listed in table 1.5. The instruction examples given in the table assume

that a temporary register called `temp` has been declared. All these instructions can be emulated using one or more instructions of NV_vertex_program.

Table 1.5: New Instructions in ARB_vertex_program

Opcode	Arity	Description	Example
FLR	Unary	Performs a component-wise floor operation on the source vector.	`FLR temp, temp;`
FRC	Unary	Computes the component-wise fractional portion of the source vector.	`FRC temp, vertex.color;`
EX2	Unary	Computes an approximation (that has higher precision as the EXP instruction) of the base 2 raised to the power of the given source scalar and replicates the result to all four components of the destination register.	`EX2 temp, vertex.position.x;`
LG2	Unary	Computes an approximation (that has higher precision as the LOG instruction) of the base 2 logarithm of the source scalar and replicates the result to all four components of the destination register.	`LG2 temp, vertex.position.z;`
XPD	Binary	Computes the three-component vector cross product of the two given source vectors.	`XPD temp, temp, vertex.normal;`
SWZ	Unary	Performs an extended swizzle operation on the source vector. The extended swizzle can not only swap or replicate components of the source vector, but also set components to either 0 or 1, if desired.	`SWZ temp, temp, 1, 0, y, z;`
POW	Binary	Raises the first source scalar to the power of the second source scalar and replicates the result to all four destination register components.	`POW temp, verex.attrib[0].x, verex.attrib[1].y;`

1.3.4 Direct3D 9 Shading Languages

With the release of DirectX 9.0 [Micr02] in December 2002, Microsoft also introduced new versions of the vertex and pixel assembly shading languages of Direct3D. In particular the language versions 2.0, 3.0, and, at the last moment to accommodate the features of NVIDIA's GeForce FX graphics chip, the so-called version 2.x or 2.0 Extended were added for both vertex and pixel shaders. The most important new features are a higher number of instruction slots and a couple of new instructions, most notably for flow control. In addition to the new versions of the low-level vertex and fragment shading languages, Direct3D 9 also has a high-level shading language called HLSL (High-Level Shader Language). It can be considered syntactically and semantically equivalent to NVIDIA's high-level shading language Cg and will be discussed in section 1.4.2.

1.3.4.1 Direct3D 9 Vertex Shader Assembler

The vertex shading assembly language of Direct3D 9 has improved in various areas. The number of parameter registers has increased from 96 to 256. There are new boolean registers used for conditional execution and new integer registers used as counters in loop and repeat blocks. For language version 2.0 the maximum number of instructions has been pushed up to 256, for version 3.0 even to 512 or possibly more (depending on the used hardware). A couple of new arithmetic instructions have been added, such as an

instruction for computing the vector cross product, or a power instruction. These instructions are basically the same as the ones introduced with the ARB_vertex_program OpenGL extension and can be considered macro instructions, since they could be emulated using multiple instructions before. Also an instruction called `sincos` to compute the sine and cosine of a value and an instruction `nrm` to normalize a vector have been added. The really interesting new instructions in version 2.0, however, are the instructions for static flow control listed in table 1.6.

Table 1.6: Direct3D 9 Vertex Shading Language Static Flow Control Instructions

Opcode	Description	Example
label	Marks the next instruction as having a label index. A label defines a position in the vertex shader that other flow control instructions use to jump to.	`label l1`
call	Performs a function call to the given label index.	`call l1`
callnz	Performs a conditional function call to the given label, if a given boolean register is not zero.	`callnz l1, b2`
ret	Returns from a subroutine. Multiple return statements are not permitted in a subroutine.	`ret`

rep	Starts a repeat block that loops according to the repeat count specified in the given integer register. Repeat loops cannot be nested.	`rep i0`
		`add r0, r1`
		`endrep`

endrep	Ends a repeat block started with the rep instruction	

loop	Starts a loop block. A loop starts from a specified initial value with a specified iteration count and increment. These values are specified in a given integer register. The current loop count is stored in the loop counter register called aL. Loops can be nested in versions 2.x and above.	`loop aL, i2`
		`add r0, c[aL]`
		`endloop`

endloop	Ends a loop block.	

if	Starts an if block. If the given boolean register is true, the code enclosed by the if and the matching else instruction is run. Otherwise, the code enclosed by the else and endif instruction is run. If blocks can be nested.	`if b0`
		`mov r0, r1`
		`else`
		`mov r0, r2`
		`endif`

else	Starts an else block for a preceeding if block.	
endif	Ends an if block.	

With these instructions it is possible to use if-statements, write loops, and call subroutines in vertex shaders. This is useful in situations where a lot of code had to be duplicated in older versions of the language, for example when computing per-vertex lighting for more than one light source for a scene where the computations required for each light source are the same.

Versions 2.x and 3.0 of the language add dynamic flow control instructions that allow if-statements, subroutine calls and loops that only get executed based on a condition depending on values computed earlier in the shader. Also breaking out of loops, again possibly with a specific condition, is possible with new `break` instructions.

Finally, version 3.0 adds extended relative addressing into more register banks and so-called vertex textures. In previous versions only the parameter registers could be indexed using relative addressing. In version 3.0, also the input and output registers can be indexed with the loop counter register. Vertex textures are textures that can be sampled in the vertex shader by the use of a special texture addressing instruction. The vertex shader has access to new texture stages that are independent of the texture stages at the fragment level. Vertex textures are a powerful feature that gives vertex shaders easy access to large memory chunks. Currently, no available graphics hardware supports vertex textures or any other features of language version 3.0, though.

1.3.4.2 Direct3D 9 Pixel Shader Assembler

The programmable fragment processing stage is probably the biggest improvement in Direct3D 9 over its predecessor Direct3D 8 (apart from the addition of a high-level shading language). The most changes and improvements were made to the pixel shading assembly language. Most importantly, the number of available instructions has been increased to 64 arithmetic instructions and 32 texture instructions for language version 2.0 and even 512 minimum or more for version 3.0.

Texture coordinate and texture sampler registers have been completely separated into two different register banks. t0 to t7 are the 8 texture coordinate registers, and s0 to s15 are the 16 sampler registers that identify a texture sampling stage. For performing a texture lookup only three texture instructions are available, significantly cutting down the unnecessarily high number of texture addressing instructions of previous language versions (see section 1.3.1.2). The three texture instructions are: `texld` for regularly sampling a texture by the use of a specific texture sampler stage and a set of texture coordinates, `texldp` for projective texture sampling, and `texldb` for texture sampling with a mipmap level of detail bias.

With version 2.0 and up the pixel shading language now supports all the arithmetic instructions that are also supported by the vertex shading language, which allows very powerful fragment shaders. Even the `log`, `exp` and `sincos` instructions are supported at the fragment level. Also the `nrm` instruction for normalizing a vector is available, no longer making the use of normalization cube maps to normalize vectors in a fragment shader necessary. However, version 2.0 does not yet support any kind of flow control, not even static flow control.

Flow control is introduced in language versions 2.x and 3.0. All the static flow control instructions of the vertex shading language version 2.0, such as `call`, `if`, `rep`, and `loop` (see table 1.6), are available with version 2.x. Additionally, the dynamic flow control instructions, just as in the corresponding versions of the vertex shading language, have been added. Finally, instructions to compute the partial derivatives relative to the x and y window coordinates of a fragment have been introduced. For language version 2.x all these features depend on certain capability flags that are set depending on whether the hardware supports a particular feature. In other words, with version 2.x all these features are optional. In version 3.0 they must, however, be supported. At present, no hardware available on the market supports language version 3.0 or 2.x with flow control instructions.

1.3.5 NV_vertex_program2

The NV_vertex_program2 OpenGL extension [Kilg02b][Kilg02c] of NVIDIA introduces an extended execution environment to the NV_vertex_program extension. It is currently only available on NVIDIA's newest generation GPU GeForce FX. This new extension offers a number of new, very powerful instructions, such as dynamic branching, looping and subroutine calls. Also sine and cosine, high-precision exponentiation and logarithm, and a couple of other convenient instructions have been added, which can, however, for the most part be emulated by using multiple instructions in earlier versions of the language. The maximum number of instructions per shader has been doubled to 256. The number of parameter registers has been increased from 96 to 256.

Feature-wise, the extension corresponds to the Direct3D 9 vertex shading language version 2.x discussed in the previous section. Syntax-wise the languages are slightly different, though. For example, labels in NV_vertex_program2 are declared by specifying an identifier followed by a colon, whereas in the Direct3D 9 vertex shading language labels are declared using the pseudo-instruction `label`. Furthermore, in Direct3D 9 only forward calls are allowed, that is, a label must be declared after all branch or call instructions that reference that label. NV_vertex_program2 does not have any such restriction. Apart from these syntactical differences, both languages are computationally equally powerful.

1.3.6 ARB_fragment_program

The ARB_fragment_program OpenGL extension [Brow02b] is the first fragment shader extension approved by the ARB and is the fragment-level counterpart to ARB_vertex_program. It uses the same function entry points to upload fragment shaders to the hardware and to set parameters as ARB_vertex_program, and defines a fragment shading assembly language that is almost as powerful as its vertex-level counterpart. The language supports sine and cosine instructions, as well as exponentiation and logarithm computation instructions. Unlike previous fragment shading languages it also supports full-featured operand component swizzling, as defined in ARB_vertex_program. Just as in the Direct3D 9 fragment shading language, three texture fetching instructions are available: `TEX` is used to regularly sample textures, `TEXP` is used to perform projective texture mapping, and `TEXB` performs texture mapping with a mipmap level of detail bias. Also a `KIL` instruction is available to prevent a fragment from being passed on to the subsequent stages of the graphics pipeline.

ARB_fragment_program is equally powerful as the Direct3D 9 fragment shading language version 2.0, but not as powerful as version 2.x or 3.0 because of the lack of instructions for static or dynamic flow control. Thus branching, subroutine calls and loops are not supported. A future extension of the language is very likely to provide these features, though, when hardware becomes available that offers flow control in the programmable fragment processing stage.

1.3.7 NV_fragment_program

NV_fragment_program [Kilg02b][Kilg02c] is an NVIDIA-proprietary fragment shader OpenGL extension that basically corresponds to the Direct3D 9 fragment shading language version 2.x, but is a bit more powerful in certain areas in that it slightly lifts some restrictions, and a bit less powerful regarding flow control instructions, which it does not support. Fragment shaders can have a maximum of 1024 instructions instead of the maximum 512 instructions in Direct3D 9. Also NV_fragment_program can execute instructions at different levels of precision, if desired. Arithmetic instructions can be performed at either 32-bit floating point precision, 16-bit floating point precision, or 12 bit fixed point precision. The precision of individual instructions is specified by adding a one letter suffix representing the various levels of precision to the instruction opcode.

NV_fragment_program does not offer static or dynamic flow control instructions, but, thanks to a special condition code register, allows the construction of if-statements. This is achieved by executing both the if- and the else-block of the statement storing the results in temporary registers. Then the condition gets evaluated, thus setting the condition code register. Depending on the result, one of the previously computed temporary values is chosen. Just as ARB_fragment_program and the Direct3D 9 fragment shading language version 2.x, NV_fragment_program has instructions to compute the sine, cosine, exponential and logarithm of a value, and additionally provides instructions to compute approximate partial derivatives with respect to the x and y window coordinates. Furthermore, NV_fragment_program has pack and unpack instructions with which it is possible to pack and then unpack four 8 bit scalars into 32 bit floating point registers. This is useful for storing multiple channels in a single destination buffer and is mostly used in the process of rendering to a floating point texture. Considering its features and the fact that NV_fragment_program is available in hardware in the form of the GeForce FX GPU, it is the most powerful fragment shading language implemented in graphics hardware currently available.

1.4 High-Level Shading Languages

With low-level shading languages becoming more and more powerful and thus also more complex, and due to the variety of available assembly-like languages, the need for high-level shading languages for graphics programming became apparent. Similar to the move from assembly languages to high-level programming languages in the area of general purpose CPU programming, high-level shading languages are beginning to emerge that abstract from the assembly-like languages predominant until recently.

Syntax-wise, most of the high-level shading languages available today are based on the programming language C and tus are structured languages. The syntax for flow control statements and functions is just as in C. However, the supported data types are very limited. The integral data types usually include a IEEE 32-bit floating-point type and vector and matrix types which are typically useful in graphics programming. Integer and boolean data types are also available in most languages. String or character data types are not supported in any language at the moment.

The following sections discuss three high-level, real-time shading languages that can be used on consumer graphics hardware. First, the Stanford Real-Time Shading Language is introduced, which served as scientific basis for the other languages presented. Then NVIDIA's Cg is presented, which was the first high-level language to gain wide popularity. The discussion of Cg equally applies to the Direct3D 9.0 HLSL, which is syntactically and semantically equivalent to Cg. Finally, the current draft of the Glslang shading language is discussed. Glslang will be released as the official high-level shading language of OpenGL 2.0 and is currently still under development by the corresponding ARB working group.

1.4.1 Stanford Real-Time Shading Language

The *Stanford Real-Time Shading Language* [Prou00][Mark01][Prou01] was the first real-time shading language specifically designed for programmable consumer hardware (unlike Olano's work with the PixelFlow system [Olan98][Eyle97]). As opposed to the other shading languages presented in this paper, the Stanford language does not distinguish between vertex and fragment shaders, but rather combines them into a single so-called surface shader. Surface shaders return a four-component RGBA colour as final output that gets passed on as fragment colour to the final pipeline stages. Furthermore so-called light shaders can be written that perform luminance calculations for lights. Light shaders can only be used from surface shaders and return a four-component RGBA light colour.

A surface shader can be seen as a program for the entire pipeline and not just a single pipeline stage as with other shading languages. Therefore, the Stanford Shading Language is not as hardware-centric as most other real-time shading languages. However, the execution environment does not differ significantly from what has been discussed so far. The compiler internally breaks down the shader to multiple shader blocks that each program a specific pipeline stage, as can be seen in the abstraction of the programmable pipeline for the Stanford system in figure 1.4. The figure only shows the programmable stages of the pipeline. These stages are connected by fixed-function stages that convert between computation frequencies just as in figure 1.1. Note that the application can only directly pass in data to the primitive group and the vertex processing stages.

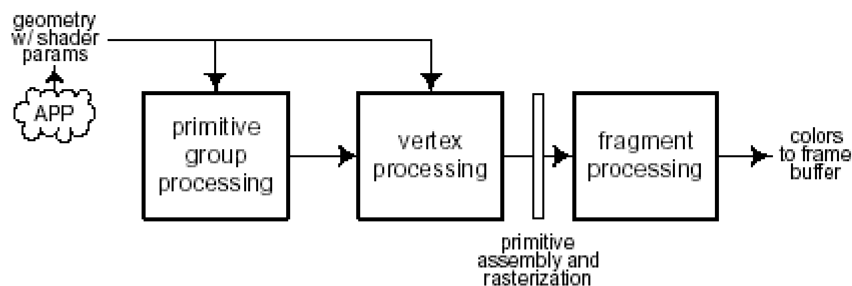

Figure 1.4: Stanford Programmable Pipeline Abstraction

In addition to the two computation frequencies, per-vertex and per-fragment, the Stanford language also has the concept of a constant and a per-primitive group computation

frequency as shown in figure 1.5. Constant computations are evaluated by the compiler at compile time. Other shading languages, such as Cg or Glslang, offer this as well, but do not regard it as separate computation frequency, but rather as compiler optimization. Per-primitive group computations influence values that do not change for a number of primitives. For example, they compute a new projection matrix. The Stanford language is the only shading language that supports per-primitive group computations in the shading language itself. For all other shading languages presented in this paper per-primitive computations must be done on the CPU in the general purpose programming language that is used to develop the main application. The results of these computations are then bound to parameter registers to give the shader access to them. Since no graphics hardware currently supports per-primitive group computations, the Stanford compiler actually compiles per-primitive group shader code to machine code of the host CPU.

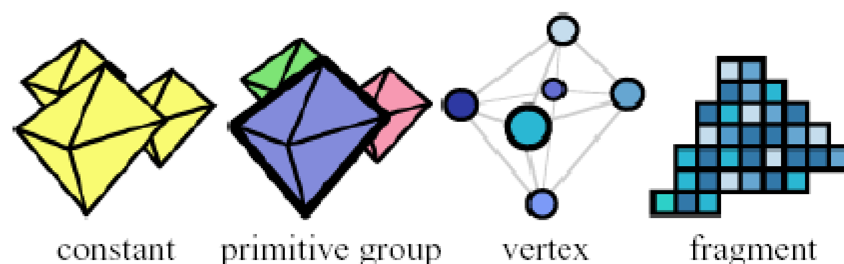

Figure 1.5: Computation Frequencies in the Stanford System

The Stanford Shading Language itself is loosely based on the RenderMan Shading Language [Hanr90] omitting features that were not possible on consumer graphics hardware at the time the system was devised, such as loops and conditionals. In addition to the base data type of the RenderMan Shading Language, float, the Stanford language adds other data types that are useful in the context of real-time shading languages for programmable graphics hardware. In particular, the Stanford language supports ten data types: scalar floats, three-component and four-component vectors, each of which may be composed of either floats or floats clamped in the range [0, 1], three-by-three float matrices, four-by-four float matrices, booleans, and a special texture reference type to reference texture sampler stages when texture lookups are performed.

The operations offered by the language were chosen to support the standard transform, lighting, and texture access and blending functionality. The language offers basic scalar, vector, and matrix operations; exponentiation; square roots; dot and cross products; trigonometric functions; comparison, minimum and maximum operators; clamp operators; and type casting. Additional operations perform 2D, 3D, and, cube map texture lookups. For special-purpose complex operations that are not orthogonally supported by graphics hardware, the Stanford system furthermore offers so-called canned functions that make these operations more efficient on specific backends. In particular, the language offers two functions, `bumpdiff` and `bumpspec`, that perform per-pixel bump mapping as described by Kilgard in [Kilg00].

A unique feature of the Stanford shading system is that the shader runtime performs transparent multipassing when a shader does not fit within the hardware limits. When the compiler notices that the hardware limits have been reached, for example a shader

requires too many instructions, the shader is split up into multiple shaders and the runtime uses the render-to-texture feature of OpenGL to perform multipass rendering. This means that all passes except for the last one are rendered to a texture instead of the frame buffer, where each texture is used in subsequent passes. Naturally, due to the limited precision and blending capabilities of current texture hardware this kind of multipassing is not always possible. Also multipassing has negative effects on performance, since all geometry data must be sent to the graphics hardware multiple times.

1.4.2 Cg / Direct3D HLSL

The language *Cg* [Kirk02], short for *C for Graphics*, released by NVIDIA as a public beta in April 2002, was the first high-level shading language to find widespread use. The initial beta release contained the so-called Cg runtime, a library used to set up, manage, and compile shaders at runtime, and a command-line compiler that was able to compile to the Direct3D 8 low-level shading languages, NV_vertex_program and ARB_vertex_program. Since the initial Cg runtime did not have a convincing, well-thought-out design and suffered from a number of obvious bugs, a separate Cg runtime was developed for XEngine. Since there was no support for fragment shaders in OpenGL, either, a cross-compiler for translating Direct3D 8 pixel shaders to the corresponding OpenGL extensions NV_register_combiners and NV_texture_shader was integrated into XEngine. For a couple of months, XEngine was the only way to use fragment shaders with OpenGL and Linux thanks to that cross-compiler.

The final release of the Cg compiler and Cg runtime in December 2002 eventually got rid of most of the issues in the beta releases. The runtime was completely redesigned and rewritten, and the compiler supported compiling to the OpenGL fragment shading extensions NV_register_combiners/NV_texture_shader, and additionally to the Direct3D 9 low-level shading languages, ARB_fragment_program, and NV_fragment_program. Due to the involvement with the Cg online community and the development of a separate Cg runtime for XEngine, the author was invited by NVIDIA to be a beta tester for the new compiler and runtime.

The high-level shading language released by Microsoft with Direct3D 9 in December 2002, simply called *High-Level Shader Language* or *HLSL*, uses the same grammar as Cg. HLSL is syntactically and semantically equivalent to Cg, except that it has a different name, the runtime used to manage and compile shaders is different, and of course it

can only be used with Direct3D 9. Everything said in this section about the Cg language itself also applies to Direct3D 9's HLSL.

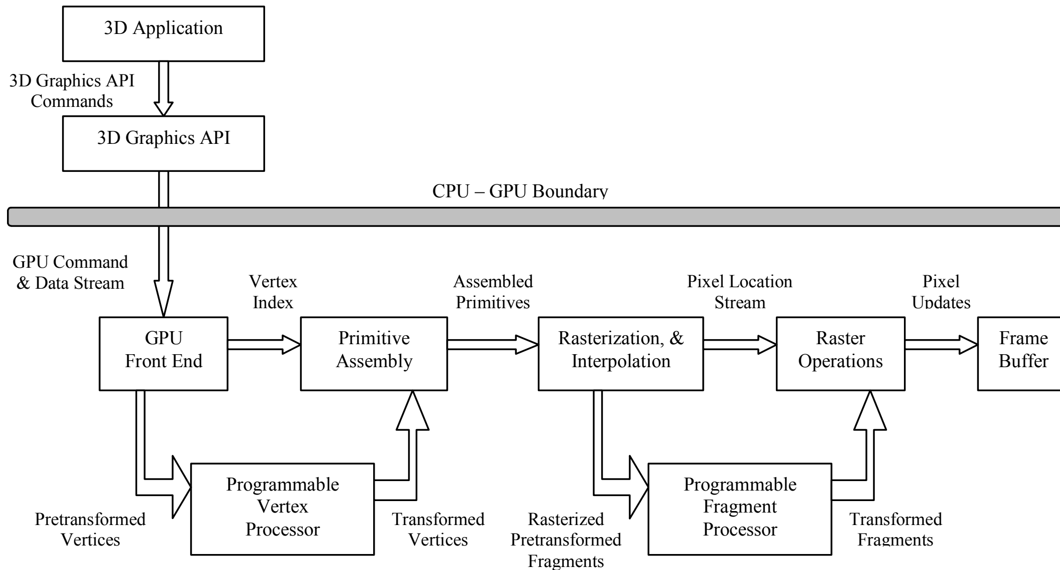

Figure 1.6: Cg's GPU Model

Cg's grammar is loosely based on the C programming language with various changes or enhancements necessary due to the fundamental differences of GPU and CPU programming. Figure 1.6 shows the GPU model used by Cg, which, not surprisingly, follows the same architecture described in section 1.2. Unlike in the Stanford Shading Language, separate programs have to be written for vertex and fragment shaders in Cg, that also need to be compiled separately. When compiling a shader, a so-called compiler *profile* must be specified. A profile defines what language features are available and what low-level shading language the compiler should use as target language. These profiles are necessary because GPU programmability has not yet reached the same generality as CPU programmability. For example, in the arbvp1 profile, the profile for ARB_vertex_program, if-statements are not allowed because ARB_vertex_program does not have instructions for flow control. The vp30 profile, which is the profile for the NV_vertex_program2 extension, on the other hand, allows if-statements and loops since that particular shading language supports dynamic flow control. It is also possible to let the Cg runtime choose an optimal profile at runtime depending on the available features of the graphics hardware.

A special language feature, the so-called *bindings* or *binding semantics*, are used to bind vertex attributes to input variables of the vertex shader, and output variables of the vertex shader to input variables of the fragment shader. The bindings represent underlying hardware registers and some of them are profile-specific, even though most profiles share most of the bindings. Bindings are specified in a variable declaration after a colon following the variable name.

Cg supports six basic data types, a 32-bit IEEE floating point type, a 16-bit IEEE-like floating point type, a 32-bit integer type, a 12-bit fixed point type, a boolean type, and special sampler types that represent handles to texture objects. Additionally, the

language supports built-in compound vector and matrix types that are based on the basic types. For example, float4 is a four-component vector type composed of four 32-bit floating point values. Furthermore, arrays and structures can be declared by using the basic and compound types, just as in the programming language C.

The statements and operators supported by Cg are largely the same as in C, except for the fact that most operations cannot only operate on scalar data types but also on compound vector and matrix types. Except for the bitwise binary operators and operators for pointers, all standard C operators are supported in Cg. Additionally, component swizzling on vector types as defined in most low-level shading languages is supported. Functions can be defined just as in C, and also function overloading similarly to C++ is allowed. Functions cannot only be overloaded by different function parameter lists, but also by different compiler profiles. In addition to the standard operators, the Cg language offers a large standard library of pre-defined functions, such as sine, cosine, dot and cross products, matrix multiplication, vector normalization, and texture fetching.

To show how convenient it can be to use a high-level shading language instead of a low-level language, the example vertex shader presented in section 1.3.1.1 that computed diffuse per-vertex lighting in world space using a single directional light is presented again, but this time written in Cg. As a note to better understand the code, in Cg, variables that do not change per-vertex, such as the combined world-view-projection matrix, are declared with the `uniform` keyword. These uniform variables are assigned to parameter registers by the compiler and set by the application using the Cg runtime.

```
struct Output
{
     float4 pos : POSITION;
     float4 color : COLOR0;
     float2 texcoord : TEXCOORD0;
};

Output main(
          float4 pos : POSITION,
          float3 normal : NORMAL,
          float2 texcoord : TEXCOORD0,
          uniform float4x4 worldViewProj,
          uniform float4x4 invTransWorld,
          uniform float3 lightDir,
          uniform float4 diffuseColor,
          uniform float4 ambientColor)
{
     Output output;

     // transform the vertex position to homogeneous clip space
     output.pos = mul(worldViewProj, pos);

     // transform the normal from local to world space
     float3 worldNormal = mul((float3x3)invTransWorld, normal);

     // normalize the normal and the light vector
     worldNormal = normalize(worldNormal);
     float3 worldLightDir = normalize(lightDir);
```

```
        // perform the lighting computation
        float diffuse = max(0, dot(worldNormal, worldLightDir));
        output.color = ambientColor + diffuse * diffuseColor;

        // just pass through the texture coordinates
        output.texcoord = texcoord;

        return output;
}
```

This shader is easier to read than the low-level shader presented earlier. Especially for more complicated shaders, using a high-level shading language can drastically shorten the development cycle.

1.4.3 Glslang

Glslang, short for *GL Shading Language*, is the tentative name of the standardized high-level shading language of OpenGL, which will either be introduced with OpenGL 2.0 or, more likely, earlier in the form of regular OpenGL extensions that will later be integrated into core OpenGL 2.0. At the time of this paper being written, the language itself has not yet been finalized. The corresponding ARB working group is still working on the language specification. Glslang was first introduced in draft papers presented by 3Dlabs, which is what the following discussion is based on [Bald03][Rost02]. Glslang is nevertheless presented here, even though it is not yet finalized, because it can be expected to be one of the most important real-time, high-level shading languages once it becomes part of core OpenGL. Just as Cg, Glslang is largely based on the C programming language and is generally very similar to Cg in a lot of respects. Glslang has the same GPU model as Cg and also requires separate programs to be written for vertex and fragment shaders.

Glslang has ten basic data types, a boolean type, signed and unsigned integer types, one-, two-, three-, and four-component 32-bit floating-point types, and a two-by-two, three-by-three, and four-by-four 32-bit floating point matrix type. Arrays and structures of these basic types can be declared. All the standard operators offered by C are supported by Glslang, except for operators dealing with pointers since pointer types are not part of the shading language. Additionally, component swizzling on the vector types is also allowed. The statements supported by Glslang are the same as Cg, so everything from an if-statement to a for-loop is supported. Functions are declared just as in C, and just as in Cg, C++-like function overloading is also permitted. The built-in functions offered by Glslang are mostly the same as Cg. Functions for normalizing vectors, computing the sine and cosine of a value, multiplying matrices, and many other operations are provided.

Instead of using bindings to specify input and output variables to the shaders, Glslang uses pre-defined, global, read-only and write-only variables that the shaders use to access vertex attributes, fragment shader inputs, or GL state. For example, a vertex shader reads the input vertex position from the global read-only variable gl_Vertex and writes the computed homogeneous clip position of the vertex to the global variable

gl_Position. The fragment shader then receives the position interpolated by the rasterizer in the global read-only variable gl_FragCoord.

1.5 Conclusion

GPU programmability has increased rapidly in the last couple of years and will probably continue to do so for quite a while. The development undergone by shading languages has roughly been the same as for general-purpose programming languages so far. The first shading languages were primitive assembly languages with a very limited instruction set. The instruction sets grew to include flow control instructions as well, and procedural high-level languages began to emerge. It is only a matter of time until object-oriented principles are integrated into new high-level shading languages. The seemingly unreachable goal of being able to use the RenderMan Shading Language, which was devised as non-real-time shading language for ray tracing, in real-time on consumer graphics hardware has come into reachable grasp within the last few years. Judging from the pace of evolvement of real-time shading languages, this goal will become reality within the next two or three generations of graphics hardware.

To further underline the importance of real-time shading, the first development environments designed exclusively for shader development have been released, and also traditional ray tracing programs have begun to use real-time shading for preview purposes. Since entertainment companies are already hiring people whose job it is to exclusively develop real-time shaders, it is a definite plus for a graphics software engineer to be familiar with some of the shading languages presented.

Bibliography

[Bald03] Dave Baldwin, Randi J. Rost, John Kessenich: *The OpenGL Shading Language, Version 1.05*, 3Dlabs Inc, 2003

[Brow02a] Pat Brown, et al.: *ARB_vertex_program OpenGL Extension Specification*, ARB, 2002

[Brow02b] Pat Brown, et al.: *ARB_fragment_program OpenGL Extension Specification*, ARB, 2002

[Eyle97] John Eyles, et al.: *PixelFlow: The Realization*, Hewlett-Packard Company, Chapel Hill Graphics Labs, University of North Carolina, Department of Computer Science, 1997

[Hanr90] Pat Hanrahan, Jim Lawson: *A Language for Shading and Lighting Calculations*, ACM SIGGRAPH Computer Graphics, Volume 24, Number 4, 1990

[Kilg00] Mark J. Kilgard: *A Practical and Robust Bump-mapping Technique for Today's GPUs*, Games Developers Conference 2000: Advanced OpenGL Game Development, NVIDIA Corporation, 2000

[Kilg02a] Mark J. Kilgard, et al.: *NVIDIA OpenGL Extension Specifications*, Editor: Mark J. Kilgard, NVIDIA Corporation, 2003

[Kilg02b] Mark J. Kilgard, et al.: *NVIDIA OpenGL Extension Specifications for the CineFX Architecture (NV30)*, Editor: Mark J. Kilgard, NVIDIA Corporation, 2003

[Kilg02c] Mark J. Kilgard: *NV30 OpenGL Extensions*, Presentation, NVIDIA Corporation, 2002

[Kirk02] David Kirk, et al.: *Cg Toolkit User's Manual. A Developer's Guide to Programmable Graphics (Release 1.0)*, NVIDIA Corporation, 2002

[Lind00a] Erik Lindholm: *Vertex Program Modules*, NVIDIA Corporation, 2000

[Lind00b] Erik Lindholm: *Vertex Programs for Fixed Function*, NVIDIA Corporation, 2000

[Mark01] William R. Mark, Kekoa Proudfoot: *Compiling to a VLIW Fragment Pipeline*, in Proceedings of 2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware, Stanford University, Department of Computer Science, 2001

[Micr01] n.n.: *Microsoft DirectX 8.1 Programmer's Reference*, DirectX 8.1 SDK, Microsoft Corporation, 2001

[Micr02] n.n.: *Microsoft DirectX 9.0 Programmer's Reference*, DirectX 9.0 SDK, Microsoft Corporation, 2001

[Olan98] Marc Olano: *A Programmable Pipeline for Graphics Hardware*, PhD. Thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 1998

[Prou00] Kekoa Proudfoot: *Version 5 Real-Time Shading Language Description*, Stanford University, Department of Computer Science, 2000

[Prou01] Kekoa Proudfoot, William R. Mark, et al.: *A Real-Time Procedural Shading System for Programmable Graphics Hardware*, ACM SIGGRAPH 2001, pp. 159-170, 2001

[Rost02] Randi J. Rost, Barthold Lichtenbelt, et al.: *OpenGL 2.0 White Papers*, 3Dlabs Inc, 2002

[Wlok01] Matthias Wloka: *Where Is That Instruction? How to Implement "Missing" Vertex Shader Instructions*, NVIDIA Corporation, 2001