Appendix A
Cg Language Specification

Language Overview

The Cg language is primarily modeled on ANSI C, but adopts some ideas from modern languages such as C++ and Java, and from earlier shading languages such as RenderMan and the Stanford shading language. The language also introduces a few new ideas. In particular, it includes features designed to represent data flow in stream-processing architectures such as GPUs. Profiles, which are specified at compile time, may subset certain features of the language, including the ability to implement loops and the precision at which certain computations are performed.

Silent Incompatibilities

Most of the changes from ANSI C are either omissions or additions, but there are a few potentially *silent incompatibilities*. These are changes within Cg that could cause a program that compiles without errors to behave in a manner different from C:

- The type promotion rules for constants are different when the constant is not explicitly typed using a type cast or type suffix. In general, a binary operation between a constant that is not explicitly typed and a variable is performed at the variable's precision, rather than at the constant's default precision.
- Declarations of **`struct`** perform an automatic **`typedef`** (as in C++) and thus could override a previously declared type.
- Arrays are first-class types that are distinct from pointers. As a result, array assignments semantically perform a copy operation for the entire array.

Similar Operations That Must be Expressed Differently

There are several changes that force the same operation to be expressed differently in Cg than in C:

- A Boolean type, **`bool`**, is introduced, with corresponding implications for operators and control constructs.

- Arrays are first-class types because Cg does not support pointers.
- Functions pass values by value/result, and thus use an **out** or **inout** modifier in the formal parameter list to return a parameter. By default, formal parameters are **in**, but it is acceptable to specify this explicitly. Parameters can also be specified as **in out**, which is semantically the same as **inout**.

Differences from ANSI C

Cg was developed based on the ANSI-C language with the following major additions, deletions, and changes. (This is a summary—more detail is provided later in this document):

- Language profiles (described in "Profiles" on page 168) may subset language capabilities in a variety of ways. In particular, language profiles may restrict the use of **for** and **while** loops. For example, some profiles may only support loops that can be fully unrolled at compile time.
- A *binding semantic* may be associated with a structure tag, a variable, or a structure element to denote that object's mapping to a specific hardware or API resource. See "Binding Semantics" on page 183.
- Reserved keywords **goto**, **break**, and **continue** are not supported.
- Reserved keywords **switch**, **case**, and **default** are not supported. Labels are not supported either.
- Pointers and pointer-related capabilities (such as the **&** and **->** operators) are not supported.
- Arrays are supported, but with some limitations on size and dimensionality. Restrictions on the use of computed subscripts are also permitted. Arrays may be designated as **packed**. The operations allowed on packed arrays may be different from those allowed on unpacked arrays. Predefined **packed** types are provided for vectors and matrices. It is strongly recommended these predefined types be used.
- There is a built-in swizzle operator: **.xyzw** or **.rgba** for vectors. This operator allows the components of a vector to be rearranged and also replicated. It also allows the creation of a vector from a scalar.
- For an lvalue, the swizzle operator allows components of a vector or matrix to be selectively written.
- There is a similar built-in swizzle operator for matrices:

  ._m<*row*><*col*>[_m<*row*><*col*>][…]

  This operator allows access to individual matrix components and allows the creation of a vector from elements of a matrix. For compatibility with

DirectX 8 notation, there is a second form of matrix swizzle, which is described later.

- Numeric data types are different. Cg's primary numeric data types are **`float`**, **`half`**, and **`fixed`**. Fragment profiles are required to support all three data types, but may choose to implement **`half`** and **`fixed`** at **`float`** precision. Vertex profiles are required to support **`half`** and **`float`**, but may choose to implement **`half`** at **`float`** precision. Vertex profiles may omit support for **`fixed`** operations, but must still support definition of **`fixed`** variables. Cg allows profiles to omit run-time support for **`int`**. Cg allows profiles to treat **`double`** as **`float`**.
- Many operators support per-element vector operations.
- The **`?:`**, **`||`**, **`&&`**, **`!`**, and comparison operators can be used with **`bool`** four-vectors to perform four conditional operations simultaneously. The side effects of all operands to the **`?:`**, **`||`**, and **`&&`** operators are always executed.
- Non-static global variables and parameters to top-level functions—such as **`main()`**—may be designated as **`uniform`**. A **`uniform`** variable may be read and written within a program, just like any other variable. However, the **`uniform`** modifier indicates that the initial value of the variable or parameter is expected to be constant across a large number of invocations of the program.
- A new set of **`sampler*`** types represents handles to texture objects.
- Functions may have default values for their parameters, as in C++. These defaults are expressed using assignment syntax.
- Function overloading is supported.
- There is no **`enum`** or **`union`**.
- Bit-field declarations in structures are not allowed.
- There are no bit-field declarations in structures.
- Variables may be defined anywhere before they are used, rather than just at the beginning of a scope as in C. (That is, we adopt the C++ rules that govern where variable declarations are allowed.) Variables may not be redeclared within the same scope.
- Vector constructors, such as the form **`float4(1,2,3,4)`**, may be used anywhere in an expression.
- A **`struct`** definition automatically performs a corresponding **`typedef`**, as in C++.
- C++-style **`//`** comments are allowed in addition to C-style **`/*…*/`** comments.

Detailed Language Specification

Definitions

The following definitions are based on the ANSI C standard:

- *Object*
  An object is a region of data storage in the execution environment, the contents of which can represent values. When referenced, an object may be interpreted as having a particular type.
- *Declaration*
  A declaration specifies the interpretation and attributes of a set of identifiers.
- *Definition*
  A declaration that also causes storage to be reserved for an object or code that will be generated for a function named by an identifier is a definition.

Profiles

Compilation of a Cg program, a top-level function, always occurs in the context of a compilation profile. The profile specifies whether certain optional language features are supported. These optional language features include certain control constructs and standard library functions. The compilation profile also defines the precision of the **`float`**, **`half`**, and **`fixed`** data types, and specifies whether the **`fixed`** and **`sampler*`** data types are fully or only partially supported. The choice of a compilation profile is made externally to the language, by using a compiler command-line switch, for example.

The profile restrictions are only applied to the top-level function that is being compiled and to any variables or functions that it references, either directly or indirectly. If a function is present in the source code, but not called directly or indirectly by the top-level function, it is free to use capabilities that are not supported by the current profile.

The intent of these rules is to allow a single Cg source file to contain many different top-level functions that are targeted at different profiles. The core Cg language specification is sufficiently complete to allow all of these functions to be parsed. The restrictions provided by a compilation profile are only needed for code generation, and are therefore only applied to those functions for which code is being generated. This specification uses the word *program* to refer to the top-level function, any functions the top-level function calls, and any global variables or **`typedef`** definitions it references.

Each profile must have a separate specification that describes its characteristics and limitations.

This core Cg specification requires certain minimum capabilities for all profiles. In some cases, the core specification distinguishes between vertex-program and fragment-program profiles, with different minimum capabilities for each.

The Uniform Modifier

Non-static global variables and parameters passed to functions, such as **`main()`**, can be declared with an optional qualifier **`uniform`**. To specify a **`uniform`** variable, use this syntax:

`uniform <type> <variable>`

For example,

```
uniform float4 myVector;
```

or

```
fragout foo(uniform float4 uv);
```

If the **`uniform`** qualifier is specified for a function that is not top level, it is meaningless and is ignored. The intent of this rule is to allow a function to serve either as a top-level function or as one that is not.

Note that **`uniform`** variables may be read and written just like non-**`uniform`** variables. The **`uniform`** qualifier simply provides information about how the initial value of the variable is to be specified and stored, through a mechanism external to the language.

Typically, the initial value of a **`uniform`** variable or parameter is stored in a different class of hardware register. Furthermore, the external mechanism for specifying the initial value of **`uniform`** variables or parameters may be different than that used for specifying the initial value of non-**`uniform`** variables or parameters. Parameters qualified as **`uniform`** are normally treated as persistent state, while non-**`uniform`** parameters are treated as streaming data, with a new value specified for each stream record (such as within a vertex array).

Function Declarations

Functions are declared essentially as in C. A function that does not return a value must be declared with a **`void`** return type. A function that takes no parameters may be declared in one of two ways:

- As in C, using the **`void`** keyword: **`functionName(void)`**
- With no parameters at all: **`functionName()`**

Functions may be declared as **static**. If so, they may not be compiled as a program and are not visible from other compilation units.

Overloading of Functions by Profile

Cg supports overloading of functions by compilation profile. This capability allows a function to be implemented differently for different profiles. It is also useful because different profiles may support different subsets of the language capabilities, and because the most efficient implementation of a function may be different for different profiles.

The profile name must immediately precede the type name in the function declaration. For example, to define two different versions of the function **myfunc()** for the **profileA** and **profileB** profiles:

```
profileA float myfunc(float x) {/*...*/};
profileB float myfunc(float x) {/*...*/};
```

If a type is defined (using a **typedef**) that has the same name as a profile, the identifier is treated as a type name and is not available for profile overloading at any subsequent point in the file.

If a function definition does not include a profile, the function is referred to as an *open-profile* function. Open-profile functions apply to all profiles.

Several wildcard profile names are defined. The name **vs** matches any vertex profile, while the name **ps** matches any fragment or pixel profile.

The names **ps_1** and **ps_2** match any DirectX 8 pixel shader 1.*x* profile or DirectX 9 pixel shader 2.*x* profile, respectively. Similarly, the names **vs_1** and **vs_2** match any DirectX vertex shader 1.*x* or 2*x*, respectively. Additional valid wildcard profile names may be defined by individual profiles.

In general, the most specific version of a function is used. More details are provided in "Function Overloading" on page 181, but roughly speaking, the search order is the following:

1. Version of the function with the exact profile overload
2. Version of the function with the most specific wildcard profile overload (such as **vs** or **ps_1**)
3. Version of the function with no profile overload

This search process allows generic versions of a function to be defined that can be overridden as needed for particular hardware.

Syntax for Parameters in Function Definitions

Functions are declared in a manner similar to C, but the parameters in function definitions may include a binding semantic (see "Binding Semantics" on page 183) and a default value.

Each parameter in a function definition takes the following form:

[uniform] <*type*> *identifier* [: <*binding_semantic*>] [= <*default*>]

where

- **<*type*>** may include the qualifiers **in**, **out**, **inout**, and **const**, as discussed in "Type Qualifiers" on page 175.
- **<*default*>** is an expression that resolves to a constant at compile time.

Default values are only permitted for **uniform** parameters, and for **in** parameters to functions that are not top-level.

Function Calls

A function call returns an rvalue. Therefore, if a function returns an array, the array may be read but not written. For example, the following is allowed:

```
y = myfunc(x)[2];
```

But, this is not: **myfunc(x)[2] = y;**.

For multiple function calls within an expression, the calls can occur in *any* order— it is undefined.

Types

Cg's types are as follows:

- The **int** type is preferably 32-bit two's complement. Profiles may optionally treat **int** as **float**.
- The **float** type is as close as possible to the IEEE single precision (32-bit) floating point. Profiles must support the **float** data type.
- The **half** type is lower-precision IEEE-like floating point. Profiles must support the **half** type, but may choose to implement it with the same precision as the **float** type.
- The **fixed** type is a signed type with a range of at least [-2,2) and with at least 10 bits of fractional precision. Overflow operations on the data type clamp rather than wrap. Fragment profiles must support the **fixed** type, but may implement it with the same precision as the **half** or **float** types. Vertex profiles are required to provide partial support (see "Partial Support of Types" on page 173) for the **fixed** type. Vertex profiles have the option

to provide full support for the **fixed** type or to implement the **fixed** type with the same precision as the **half** or **float** types.

- The **bool** type represents Boolean values. Objects of **bool** type are either true or false.
- The **cint** type is 32-bit two's complement. This type is meaningful only at compile time; it is not possible to declare objects of type **cint**.
- The **cfloat** type is IEEE single-precision (32-bit) floating point. This type is meaningful only at compile time; it is not possible to declare objects of type **cfloat**.
- The **void** type may not be used in any expression. It may only be used as the return type of functions that do not return a value.
- The **sampler*** types are handles to texture objects. Formal parameters of a program or function may be of type **sampler***. No other definition of **sampler*** variables is permitted. A **sampler*** variable may only be used by passing it to another function as an **in** parameter. Assignment to **sampler*** variables is not permitted, and **sampler*** expressions are not permitted.

  The following **sampler*** types are always defined: **sampler**, **sampler1D**, **sampler2D**, **sampler3D**, **samplerCUBE**, and **samplerRECT**. The base **sampler** type may be used in any context in which a more specific sampler type is valid. However, a **sampler** variable must be used in a consistent way throughout the program. For example, it cannot be used in place of both a **sampler1D** and a **sampler2D** in the same program.

  Fragment profiles are required to fully support the **sampler**, **sampler1D**, **sampler2D**, **sampler3D**, and **samplerCUBE** data types. Fragment profiles are required to provide partial support (see "Partial Support of Types" on page 173) for the **samplerRECT** data type and may optionally provide full support for this data type.

  Vertex profiles are required to provide partial support for the six sampler data types and may optionally provide full support for these data types.
- An **array** type is a collection of one or more elements of the same type. An **array** variable has a single index.
- Some array types may be optionally designated as *packed*, using the **packed** type modifier. The storage format of a packed type may be different from the storage format of the corresponding unpacked type. The storage format of packed types is implementation dependent, but must be consistent for any particular combination of compiler and profile. The operations supported on a packed type in a particular profile may be different than the operations supported on the corresponding unpacked type in that same profile. Profiles may define a maximum allowable size for packed arrays, but must support at least size 4 for packed vector (one-dimensional array) types, and 4x4 for packed matrix (two-dimensional array) types.

- When declaring an array of arrays in a single declaration, the **packed** modifier only refers to the outermost array. However, it is possible to declare a packed array of packed arrays by declaring the first level of array in a **typedef** using the **packed** keyword and then declaring a packed array of this type in a second statement. It is not possible to have a packed array of unpacked arrays.
- For any supported numeric data type ***TYPE***, implementations must support the following packed array types, which are called *vector types.* Type identifiers must be predefined for these types in the global scope:

```
typedef packed TYPE TYPE1[1];
typedef packed TYPE TYPE2[2];
typedef packed TYPE TYPE3[3];
typedef packed TYPE TYPE4[4];
```

  For example, implementations must predefine the type identifiers **float1**, **float2**, **float3**, **float4**, and so on for any other supported numeric type.

- For any supported numeric data type ***TYPE***, implementations must support the following packed array types, which are called *matrix types.* Implementations must also predefine type identifiers (in the global scope) to represent these types:

```
packed TYPE1 TYPE1x1[1];     packed TYPE1 TYPE3x1[3];
packed TYPE2 TYPE1x2[1];     packed TYPE2 TYPE3x2[3];
packed TYPE3 TYPE1x3[1];     packed TYPE3 TYPE3x3[3];
packed TYPE4 TYPE1x4[1];     packed TYPE4 TYPE3x4[3];
packed TYPE1 TYPE2x1[2];     packed TYPE1 TYPE4x1[4];
packed TYPE2 TYPE2x2[2];     packed TYPE2 TYPE4x2[4];
packed TYPE3 TYPE2x3[2];     packed TYPE3 TYPE4x3[4];
packed TYPE4 TYPE2x4[2];     packed TYPE4 TYPE4x4[4];
```

  For example, implementations must predefine the type identifiers **float2x1**, **float3x3**, **float4x4**, and so on. A **typedef** follows the usual matrix-naming convention of ***TYPE_rows_X_columns***. If we declare **float4x4 a**, then **a[3]** is equivalent to **a._m30_m31_m32_m33**.
  Both expressions extract the third row of the matrix.

- Implementations are required to support indexing of vectors and matrices with constant indices.
- A **struct** type is a collection of one or more members of possibly different types.

Partial Support of Types

This specification mandates *partial support* for some types. Partial support for a type requires the following:

- Definitions and declarations using the type are supported.
- Assignment and copy of objects of that type are supported (including implicit copies when passing function parameters).
- Top-level function parameters may be defined using that type.

If a type is partially supported, variables may be defined using that type but no useful operations can be performed on them. Partial support for types makes it easier to share data structures in code that is targeted at different profiles.

Type Categories

- The *integral* type category includes types **`cint`** and **`int`**.
- The *floating* type category includes types **`cfloat`**, **`float`**, **`half`**, and **`fixed`**. (Note that floating really means floating or fixed/fractional.)
- The *numeric* type category includes integral and floating types.
- The *compile-time* type category includes types **`cfloat`** and **`cint`**. These types are used by the compiler for constant type conversions.
- The *concrete* type category includes all types that are not included in the compile-time type category.
- The *scalar* type category includes all types in the numeric category, the **`bool`** type, and all types in the compile-time category. In this specification, a reference to a ***`<category>`*** type (such as a reference to a numeric type) means one of the types included in the category (such as **`float`**, **`half`**, or **`fixed`**).

Constants

A constant may be explicitly typed or implicitly typed. *Explicit typing* of a constant is performed, as in C, by suffixing the constant with a single character indicating the type of the constant:

- **`f`** for **`float`**
- **`d`** for **`double`**
- **`h`** for **`half`**
- **`x`** for **`fixed`**

Any constant that is not explicitly typed is *implicitly typed*. If the constant includes a decimal point, it is implicitly typed as **`cfloat`**. If it does not include a decimal point, it is implicitly typed as **`cint`**.

By default, constants are base 10. For compatibility with C, integer hexadecimal constants may be specified by prefixing the constant with **0x**, and integer octal constants may be specified by prefixing the constant with **0**.

Compile-time constant folding is preferably performed at the same precision that would be used if the operation were performed at run time. Some compilation profiles may allow some precision flexibility for the hardware; in such cases the compiler should ideally perform the constant folding at the highest hardware precision allowed for that data type in that profile.

If constant folding cannot be performed at run-time precision, it may optionally be performed using the precision indicated below for each of the numeric data types:

- **float**: s23e8 (**fp32**) IEEE single-precision floating point
- **half**: s10e5 (**fp16**) floating point with IEEE semantics
- **fixed**: s1.10 fixed point, clamping to [-2, 2)
- **double**: s52e11 (**fp64**) IEEE double-precision floating point
- **int**: signed 32-bit integer

Type Qualifiers

The type of an object may be qualified with one or more qualifiers. Qualifiers apply only to objects. Qualifiers are removed from the value of an object when used in an expression. The qualifiers are

- **const**

  The value of a **const** qualified object cannot be changed after its initial assignment. The definition of a **const** qualified object that is not a parameter must contain an initializer. Named compile-time values are inherently qualified as **const**, but an explicit qualification is also allowed.

  The value of a **static const** cannot be changed after compilation, and thus its value may be used in constant folding during compilation. A **uniform const**, on the other hand, is only **const** for a given execution of the program; its value may be changed via the runtime between executions.

- **in** and **out**

  Formal parameters may be qualified as **in**, **out**, or both (by using **in out** or **inout**). By default, formal parameters are **in** qualified. An **in** qualified parameter is equivalent to a call-by-value parameter. An **out** qualified parameter is equivalent to a call-by-result parameter, and an **inout** qualified parameter is equivalent to a value/result parameter. An **out** qualified parameter cannot be **const** qualified, nor may it have a default value.

Type Conversions

Some type conversions are allowed implicitly, while others require an cast. Some implicit conversions may cause a warning, which can be suppressed by using an explicit cast. Explicit casts are indicated using C-style syntax: casting ***variable*** to the **float4** type can be achieved using **(float4)*variable***.

- Scalar conversions

  Implicit conversion of any scalar numeric type to any other scalar numeric type is allowed. A warning may be issued if the conversion is implicit and a loss of precision is possible. Implicit conversion of any scalar object type to any compatible scalar object type is allowed. Conversions between incompatible scalar object types or between object and numeric types are not allowed, even with an explicit cast. A **sampler** is compatible with **sampler1D**, **sampler2D**, **sampler3D**, **samplerCube**, and **samplerRECT**. No other object types are compatible—**sampler1D** is not comparable with **sampler2D**, even though both are compatible with **sampler**.

  Scalar types may be implicitly converted to vectors and matrices of compatible type. The scalar is replicated to all elements of the vector or matrix. Scalar types may also be explicitly cast to structure types if the scalar type can be legally cast to every member of the structure.

- Vector conversions

  Vectors may be converted to scalar types (the first element of the vector is selected). A warning is issued if this is done implicitly. A vector may also be implicitly converted to another vector of the same size and compatible element type.

  A vector may be converted to a smaller comparable vector or a matrix of the same total size, but a warning is issued if an explicit cast is not used.

- Matrix conversions

  Matrices may be converted to a scalar type (element (0,0) is selected). As with vectors, this causes a warning if it is done implicitly. A matrix may also be converted implicitly to a matrix of the same size and shape and comparable element type.

  A matrix may be converted to a smaller matrix type (the upper right sub-matrix is selected) or to a vector of the same total size, but a warning is issued if an explicit cast is not used.

- Structure conversions

  A structure may be explicitly cast to the type of its first member or to another structure type with the same number of members, if each member of the **struct** can be converted to the corresponding member of the new **struct**. No implicit conversions of **struct** types are allowed.

- Array conversions

  No conversions of array types are allowed.

Table 6 summarizes the type conversions discussed here. The table entries have the following meanings, but please pay attention to the footnotes:

- Allowed: allowed implicitly or explicitly
- Warning: allowed, but warning issued if implicit
- Explicit: only allowed with explicit cast
- No: not allowed

Table 6 Type Conversions

	Source Type				
Target Type	**Scalar**	**Vector**	**Matrix**	**Struct**	**Array**
Scalar	Allowed	Warning	Warning	Explicit[i]	No
Vector	Allowed	Allowed[ii]	Warning[iii]	Explicit[i]	No
Matrix	Allowed	Warning[iii]	Allowed[ii]	Explicit[i]	No
Struct	Explicit	No	No	Explicit[iv]	No
Array	No	No	No	No	No

i. Only allowed if the first member of the source can be converted to the target.
ii. Not allowed if target is larger than source. **Warning** issued if target is smaller than source.
iii. Only allowed if source and target are the same total size.
iv. Only allowed if both source and target have the same number of members, and each member of the source can be converted to the corresponding member of the target.

Explicit casts are

- *Compile-time* type when applied to expressions of compile-time type
- *Numeric* type when applied to expressions of numeric or compile-time type
- *Numeric vector* type when applied to another vector type of the same number of elements
- *Numeric matrix* type when applied to another matrix type of the same number of rows and columns

Type Equivalency

Type T1 is equivalent to type T2 if any of the following are true:

- T2 is equivalent to T1.
- T1 and T2 are the same scalar, vector, or structure type.
  A packed array type is *not* equivalent to the same size unpacked array.
- T1 is a **`typedef`** name of T2.
- T1 and T2 are arrays of equivalent types with the same number of elements.
- The unqualified types of T1 and T2 are equivalent, and both types have the same qualifications.
- T1 and T2 are functions with equivalent return types, the same number of parameters, and all corresponding parameters are pair-wise equivalent.

Type-Promotion Rules

The **`cfloat`** and **`cint`** types behave like **`float`** and **`int`** types except for the usual arithmetic conversion behavior and function-overloading rules (see "Function Overloading" on page 181).

The *usual arithmetic conversions* for binary operators are defined as follows:

1. If either operand is **`double`**, the other is converted to **`double`**.
2. Otherwise, if either operand is **`float`**, the other operand is converted to **`float`**.
3. Otherwise, if either operand is **`half`**, the other operand is converted to **`half`**.
4. Otherwise, if either operand is **`fixed`**, the other operand is converted to **`fixed`**.
5. Otherwise, if either operand is **`cfloat`**, the other operand is converted to **`cfloat`**.
6. Otherwise, if either operand is **`int`**, the other operand is converted to **`int`**.
7. Otherwise, both operands have type **`cint`**.

Note that conversions happen prior to performing the operation.

Assignment

Assignment of an expression to an object or compile-time typed value converts the expression to the type of the object or value. The resulting value is then assigned to the object or value.

The value of the assignment expressions (**=**, ***=**, and so on) is defined as in C: An assignment expression has the value of the left operand after the assignment but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has a qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand occurs between the previous and the next sequence point.

Smearing of Scalars to Vectors

If a binary operator is applied to a vector and a scalar, the scalar is automatically type-promoted to a same-sized vector by replicating the scalar into each component. The ternary **?:** operator also supports smearing. The binary rule is applied to the second and third operands first, and then the binary rule is applied to this result and the first operand.

Namespaces

Just as in C, there are two namespaces. Each has multiple scopes, as in C.

- Tag namespace, which consists of **struct** tags
- Regular namespace:
  - **typedef** names (including an automatic **typedef** from a **struct** declaration)
  - Variables
  - Function names

Arrays and Subscripting

Arrays are declared as in C, except that they may optionally be declared to be **packed**, as described under “Types” on page 171. Arrays in Cg are first-class types, so array parameters to functions and programs must be declared using array syntax, rather than pointer syntax. Likewise, assignment of an **array**-typed object implies an array copy rather than a pointer copy.

Arrays with size **[1]** may be declared but are considered a different type from the corresponding non-array type.

Because the language does not currently support pointers, the storage order of arrays is only visible when an application passes parameters to a vertex or fragment program. Therefore, the compiler is currently free to allocate temporary variables as it sees fit.

The declaration and use of arrays of arrays is in the same style as in C. That is, if the 2D array **A** is declared as

```
float A[4][4];
```

then, the following statements are true:

- The array is indexed as **A[*row*][*column*]**.
- The array can be built with a constructor using

```
A =  { {A[0][0], A[0][1], A[0][2], A[0][3]},
       {A[1][0], A[1][1], A[1][2], A[1][3]},
       {A[2][0], A[2][1], A[2][2], A[2][3]},
       {A[3][0], A[3][1], A[3][2], A[3][3]} };
```

- **A[0]** is equivalent to **{A[0][0], A[0][1], A[0][2], A[0][3]}**.

Support must be provided for any **struct** containing arrays.

Minimum Array Requirements

Profiles are required to provide partial support for certain kinds of arrays. This partial support is designed to support vectors and matrices in all profiles. For vertex profiles, it is additionally designed to support arrays of light state (indexed by light number) passed as uniform parameters, and arrays of skinning matrices passed as uniform parameters.

Profiles must support subscripting, copying, and swizzling of vectors and matrices. However, subscripting with run-time computed indices is not required to be supported.

Vertex profiles must support the following operations for any non-packed array that is a uniform parameter to the program, or is an element of a structure that is a uniform parameter to the program. This requirement also applies when the array is indirectly a uniform program parameter (that is, it and or the structure containing it has been passed via a chain of **in** function parameters). The two operations that must be supported are

- Rvalue subscripting by a run-time computed value or a compile-time value
- Passing the entire array as a parameter to a function, where the corresponding formal function parameter is declared as **in**

The following operations are explicitly not required to be supported:

- Lvalue subscripting
- Copying
- Other operators, including multiply, add, compare, and so on

Note that when the array is rvalue subscripted, the result is an expression, and this expression is no longer considered to be a **`uniform`** program parameter. Therefore, if this expression is an array, its subsequent use must conform to the standard rules for array usage.

These rules are not limited to arrays of numeric types, and thus imply support for arrays of **`struct`**, arrays of matrices, and arrays of vectors when the array is a **`uniform`** program parameter. Maximum array sizes may be limited by the number of available registers or other resource limits, and compilers are permitted to issue error messages in these cases. However, profiles must support sizes of at least **`float arr[8]`**, **`float4 arr[8]`**, and **`float4x4 arr[4][4]`**.

Fragment profiles are not required to support any operations on arbitrarily sized arrays; only support for vectors and matrices is required.

Function Overloading

Multiple functions may be defined with the same name, as long as the definitions can be distinguished by unqualified parameter types and do not have an open-profile conflict (see "Overloading of Functions by Profile" on page 170).

Function-matching rules:

1. Add all visible functions with a matching name in the calling scope to the set of function candidates.
2. Eliminate functions whose profile conflicts with the current compilation profile.
3. Eliminate functions with the wrong number of formal parameters. If a candidate function has excess formal parameters, and each of the excess parameters has a default value, do not eliminate the function.
4. If the set is empty, fail.
5. For each actual parameter expression in sequence, perform the following:
    a. If the type of the actual parameter matches the unqualified type of the corresponding formal parameter in any function in the set, remove all functions whose corresponding parameter does not match exactly.
    b. If there is a defined promotion for the type of the actual parameter to the unqualified type of the formal parameter of any function, remove all functions for which this is not true from the set.
    c. If there is a valid implicit cast that converts the type of the actual parameter to the unqualified type of the formal parameter of any function, remove all functions without this cast.

d. Fail.

6. Choose a function based on profile:

   a. If there is at least one function with a profile that exactly matches the compilation profile, discard all functions that don't exactly match.

   b. Otherwise, if there is at least one function with a wildcard profile that matches the compilation profile, determine the “most specific” matching wildcard profile in the candidate set. Discard all functions except those with this most specific wildcard profile. How “specific” a given wildcard profile name is relative to a particular profile is determined by the profile specification.

7. If the number of functions remaining in the set is not one, then fail.

Global Variables

Global variables are declared and used as in C. Uniform non-static variables may have a semantic associated with them. Uniform non-static variables may have their value set through the run-time API.

Use of Uninitialized Variables

It is incorrect for a program to use an uninitialized variable. However, the compiler is not obligated to detect such errors, even if it would be possible to do so by compile-time data-flow analysis. The value obtained from reading an uninitialized variable is undefined. This same rule applies to the implicit use of a variable that occurs when it is returned by a top-level function. In particular, if a top-level function returns a **`struct`**, and some element of that **`struct`** is never written, then the value of that element is undefined.

Note: Variables are not defined as being initialized to zero because this would result in a performance penalty in cases where the compiler is unable to determine if a variable is properly initialized by the programmer.

Preprocessor

Cg profiles must support the full ANSI C standard preprocessor capabilities: **`#if`**, **`#define`**, and so on. However, Cg profiles are not required to support macro-like **`#define`** or the use of **`#include`** directives.

Overview of Binding Semantics

In stream-processing architectures, data packets flow between different programmable units. On a GPU, for example, packets of vertex data flow from the application to the vertex program.

Because packets are produced by one program (the application, in this case), and consumed by another (the vertex program), there must be some method for defining the interface between the two. Cg allows the user to choose between two different approaches to defining these interfaces.

The first approach is to associate a binding semantic with each element of the packet. This approach is a *bind-by-name* approach. For example, an output with the binding semantic **`FOO`** is fed to an input with the binding semantic **`FOO`**. Profiles may allow the user to define arbitrary identifiers in this "semantic namespace," or they may restrict the allowed identifiers to a predefined set. Often, these predefined names correspond to the names of hardware registers or API resources.

In some cases, predefined names may control non-programmable parts of the hardware. For example, vertex programs normally compute a position that is fed to the rasterizer, and this position is stored in an output with the binding semantic **`POSITION`**.

For any profile, there are two namespaces for predefined binding semantics—the namespace used for **`in`** variables and the namespace used for **`out`** variables. The primary implication of having two namespaces is that the binding semantic cannot be used to implicitly specify whether a variable is **`in`** or **`out`**.

The second approach to defining data packets is to describe the data that is present in a packet and allow the compiler decide how to store it. In Cg, the user can describe the contents of a data packet by placing all of its contents into a **`struct`**. When a **`struct`** is used in this manner, we refer to it as a *connector*. The two approaches are not mutually exclusive, as is discussed later. The connector approach allows the user to rely on a combination of user-specified semantic bindings and compiler-determined bindings.

Binding Semantics

A binding semantic may be associated with an input to a top-level function in one of three ways:

- The binding semantic is specified in the formal parameter declaration for the function. The syntax for formal parameters to a function is

  [**`const`**] [**`in | out | inout`**]
  `<type>` ***`<identifier>`*** **`[ :`** ***`<binding-semantic>`*****`][=`** ***`<initializer>`*****`]`**

- If the formal parameter is a **struct**, the binding semantic may be specified with an element of the **struct** when the **struct** is defined:

  ```
  struct <struct-tag> {
  <type> <identifier>[ : <binding-semantic>];
  /*...*/ };
  ```

- If the input to the function is implicit (a non-static global variable that is read by the function), the binding semantic may be specified when the non-static global variable is declared:

  ```
  <type> <identifier>[ : <binding-semantic>][ = <initializer>]
  ```

  If the non-static global variable is a **struct**, the binding semantic may be specified when the **struct** is defined, as described in the second bullet above.

- A binding semantic may be associated with the output of a top-level function in a similar manner:

  ```
  <type> <identifier> ( <parameter-list> )[ : <binding-semantic>]
  { <body> }
  ```

  Another method available for specifying a semantic for an output value is to return a **struct** and to specify the binding semantic(s) with elements of the **struct** when the **struct** is defined. In addition, if the output is a formal parameter, the binding semantic may be specified using the same approach used to specify binding semantics for inputs.

Aliasing of Semantics

Semantics must honor a copy-on-input and copy-on-output model. Thus, if the same input binding semantic is used for two different variables, those variables are initialized with the same value, but the variables are not aliased thereafter. Output aliasing is illegal, but implementations are not required to detect it. If the compiler does not issue an error on a program that aliases output binding semantics, the results are undefined.

Restrictions on Semantics Within a Structure

For a particular profile, it is illegal to mix input binding semantics and output binding semantics within a particular **struct**. That is, for a particular top-level function, a **struct** must be either input-only or output-only. Likewise, a **struct** must consist exclusively of uniform inputs or exclusively of non-uniform inputs. It is illegal to use binding semantics to mix the two within a single **struct**.

Additional Details for Binding Semantics

The following rules are somewhat redundant, but provide extra clarity:

- Semantics names are case-insensitive.
- Semantics attached to parameters to non-main functions are ignored.
- Input semantics may be aliased by multiple variables.
- Output semantics may not be aliased.

Using a Structure to Define Binding Semantics (Connectors)

Cg profiles may optionally allow the user to avoid the requirement that a binding semantic be specified for every non-uniform input (or output) variable to a top-level program. To avoid this requirement, all the non-uniform variables should be included within a single **`struct`**. The compiler automatically allocates the elements of this structure to hardware resources in a manner that allows any program that returns this **`struct`** to interoperate with any program that uses this **`struct`** as an input.

It is not *required* that all non-uniform inputs be included within a single **`struct`** in order to omit binding semantics. Binding semantics may be omitted from any input or output, and the compiler performs automatic allocation of that input or output to a hardware resource. However, to guarantee interoperability of one program's output with another program's input when automatic binding is performed, it is necessary to put all of the variables in a single **`struct`**.

For various reasons, it is desirable to be able to choose a particular set of rules for automatically performing the allocation of structure elements to hardware resources. The set of rules to be used, the *allocation-rule identifier*, may be specified by an identifier attached to the structure tag using the colon-identifier notation:

```
struct <struct-tag> : <allocation-rule-identifier> {
/*... */ };
```

The allocation of structure elements to hardware resources may be performed in any reproducible manner that depends only on this structure definition (and allocation-rule identifier). In particular, the allocation algorithm may not depend on how a particular function reads or writes to the elements of the structure.

Implementations may choose to make the specification of an allocation-rule identifier optional; the omission of the allocation-rule identifier implies the use of a default set of allocation rules.

It is permissible to explicitly specify a binding semantic for some elements of the **`struct`**, but not others. The compiler's automatic allocation must honor these explicit bindings. The allowed set of explicitly specified binding semantics is defined by the allocation-rule identifier. The most common use of this

capability is to bind variables to hardware registers that write to, or read from, non-programmable parts of the hardware. For example, in a typical vertex-program profile, the output **struct** would contain an element with an explicitly specified **POSITION** semantic. This element is used to control the hardware rasterizer.

For any particular allocation-rule identifier, additional restrictions may be associated with particular binding semantics. For example, it might be illegal to read from a variable with a particular binding semantic.

How Programs Receive and Return Data

A program is just a non-static function that has been designated as the main entry point at compilation time. The varying inputs to the program come from this top-level function's varying **in** parameters. The uniform inputs to the program come from the top-level function's uniform **in** parameters and from any non-static global variables that are referenced by the top-level function or by any functions that it calls. The output of the program comes from the return value of the function (which is always implicitly varying), and from any **out** parameters, which must also be varying.

Parameters to a program of type **sampler*** are implicitly **const**.

Statements

Statements are expressed just as in C, unless an exception is stated elsewhere in this document. Additionally,

- The **if**, **while**, and **for** statements require **bool** expressions in the appropriate places.
- Assignment is performed using **=**. The assignment operator returns a value, just as in C, so assignments may be chained.
- The new **discard** statement terminates execution of the program for the current data element—such as the current vertex or current fragment—and suppresses its output. Vertex profiles may choose to omit support for **discard**.

Minimum Requirements for **if**, **while**, and **for** Statements

The minimum requirements are as follows:

- All profiles should support **if**, but such support is not strictly required for older hardware.
- All profiles should support **for** and **while** loops if the number of loop iterations can be determined at compile time.

"Can be determined at compile time" is defined as follows:

> The loop-iteration expressions can be evaluated at compile time by use of intra-procedural constant propagation and folding, where the variables through which constant values are propagated do not appear as lvalues within any kind of control statement (**`if`**, **`for`**, or **`while`**) or **`?:`** construct.

Profiles may choose to support more general constant propagation techniques, but such support is not required.

- Profiles may optionally support fully general **`for`** and **`while`** loops.

New Vector Operators

These new operators are defined for vector types:

- Vector construction operator: **`<typeID>(…)`**
  This operator builds a vector from multiple scalars or shorter vectors:

```
float4(scalar, scalar, scalar, scalar)
float4(float3, scalar)
```

- Matrix construction operator: **`<typeID>(…)`**
  This operator builds a matrix from multiple rows. Each row may be specified either as multiple scalars or as any combination of scalars and vectors with the appropriate size.

```
float3x3(1, 2, 3, 4, 5, 6, 7, 8, 9)
float3x3(float3, float3, float3)
float3x3(1, float2, float3, float3, 1, 1, 1)
```

- Swizzle operator: (`.`)

```
a = b.xxyz;  // A swizzle operator example
```

  - At least one swizzle character must follow the operator.
  - There are two sets of swizzle characters and they may not be mixed. Set one is **`xyzw = 0123`**, and set two is **`rgba = 0123`**.
  - The vector swizzle operator may only be applied to vectors or to scalars.
  - Applying the vector swizzle operator to a scalar gives the same result as applying the operator to a vector of length one.
    Thus, **`myscalar.xxx`** and **`float3(myscalar,myscalar,myscalar)`** yield the same value.
  - If only one swizzle character is specified, the result is a scalar, not a vector of length one. Therefore, the expression **`b.y`** returns a scalar.

- Care is required when swizzling a constant scalar because of ambiguity in the use of the decimal point character. For example, to create a three-vector from a scalar, use one of the following:

  (1).xxx or **1..xxx** or **1.0.xxx** or **1.0f.xxx**

- The size of the returned vector is determined by the number of swizzle characters. Therefore, the size of the result may be larger or smaller than the size of the original vector.

  For example, **float2(0,1).xxyy** and **float4(0,0,1,1)** yield the same result.

- Matrix swizzle operator:

  For any matrix type of the form **<*type*><*rows*>x<*columns*>**, the notation

  <*matrixObject*>._m<*row*><*col*>[_m<*row*><*col*>][…]

  can be used to access individual matrix elements (in the case of only one **<*row*><*col*>** pair) or to construct vectors from elements of a matrix (in the case of more than one **<*row*><*col*>** pair). The row and column numbers are zero-based.

  For example,

```
float4x4 myMatrix;
float    myFloatScalar;
float4   myFloatVec4;

// Set myFloatScalar to myMatrix[3][2].
myFloatScalar = myMatrix.m_32;

// Assign the main diagonal of myMatrix to myFloatVec4.
myFloatVec4 = myMatrix.m_00_m11_m22_m33;
```

  - For compatibility with the **D3DMatrix** data type, Cg also allows one-based swizzles, using a form with the **m** omitted after the _ symbol:

    <*matrixObject*>._<*row*><*col*>[_<*row*><*col*>][…]

    In this form, the indexes for **<*row*>** and **<*col*>** are one-based, rather than the C standard zero-based. So, the two forms are functionally equivalent:

```
float4x4 myMatrix;
float4   myVec;

// These two statements are functionally equivalent:
myVec = myMatrix._m00_m23_m11_m31;
myVec = myMatrix._11_34_22_42;
```

Because of the confusion that can be caused by the one-based indexing, use of the latter notation is strongly discouraged.

- The matrix swizzles may only be applied to matrices. When multiple components are extracted from a matrix using a swizzle, the result is an appropriately sized vector. When a swizzle is used to extract a single component from a matrix, the result is a scalar.

- The write-mask operator: (`.`)
  It can only be applied to an lvalue that is a vector. It allows assignment to particular elements of a vector or matrix, leaving other elements unchanged.The only restriction is that a component cannot be repeated.

Arithmetic Precision and Range

Some hardware may not conform exactly to IEEE arithmetic rules. Fixed-point data types do not have IEEE-defined rules.

Optimizations are allowed to produce slightly different results than unoptimized code. Constant folding must be done with approximately the correct precision and range, but is not required to produce bit-exact results. It is recommended that compilers provide an option either to forbid these optimizations or to guarantee that they are made in bit-exact fashion.

Operator Precedence

Cg uses the same operator precedence as C for operators that are common between the two languages.

The swizzle and write-mask operators (`.`) have the same precedence as the structure member operator (`.`) and the array index operator (`[]`).

Operator Enhancements

The standard C arithmetic operators (`+`, `-`, `*`, `/`, `%`, `unary-`) are extended to support vectors and matrices. Sizes of vectors and matrices must be appropriately matched, according to standard mathematical rules. Scalar-to-vector promotion (see “Smearing of Scalars to Vectors” on page 179) allows relaxation of these rules.

Table 7 Expanded Operators

Operator	Description
`M[`*`n`*`][`*`m`*`]`	Matrix with ***n*** rows and ***m*** columns
`V[`*`n`*`]`	Vector with ***n*** elements

Table 7 Expanded Operators (continued)

Operator	Description
-V[*n*] -> V[*n*]	Unary vector negate
-M[*n*] -> M[*n*]	Unary matrix negate
V[*n*] * V[*n*] -> V[*n*]	Componentwise *****
V[*n*] / V[*n*] -> V[*n*]	Componentwise **/**
V[*n*] % V[*n*] -> V[*n*]	Componentwise **%**
V[*n*] + V[*n*] -> V[*n*]	Componentwise **+**
V[*n*] - V[*n*] -> V[*n*]	Componentwise **-**
M[*n*][*m*] * M[*n*][*m*] -> M[*n*][*m*]	Componentwise *****
M[*n*][*m*] / M[*n*][*m*] -> M[*n*][*m*]	Componentwise **/**
M[*n*][*m*] % M[*n*][*m*] -> M[*n*][*m*]	Componentwise **%**
M[*n*][*m*] + M[*n*][*m*] -> M[*n*][*m*]	Componentwise **+**
M[*n*][*m*] - M[*n*][*m*] -> M[*n*][*m*]	Componentwise **-**

Operators

Boolean

&& || !

Boolean operators may be applied to **bool** packed **bool** vectors, in which case they are applied in elementwise fashion to produce a result vector of the same size. Each operand must be a **bool** vector of the same size.

Both sides of **&&** and **||** are always evaluated; there is no short-circuiting as there is in C.

Comparisons

< > <= >= != ==

Comparison operators may be applied to numeric vectors. Both operands must be vectors of the same size. The comparison operation is performed in elementwise fashion to produce a **bool** vector of the same size.

Comparison operators may also be applied to **bool** vectors. For the purpose of relational comparisons, **true** is treated as one and **false** is treated as zero. The

comparison operation is performed in elementwise fashion to produce a **bool** vector of the same size.

Comparison operators may also be applied to numeric or **bool** scalars.

Arithmetic

+ - * / % ++ -- unary- unary+

The arithmetic operator **%** is the remainder operator, as in C. It may only be applied to two operands of **cint** or **int** type.

When **/** or **%** is used with **cint** or **int** operands, C rules for integer **/** and **%** apply.

The C operators that combine assignment with arithmetic operations (such as **+=**) are also supported when the corresponding arithmetic operator is supported by Cg.

Conditional Operator

?:

If the first operand is of type **bool**, one of the following statements must hold for the second and third operands:

- Both operands have compatible structure types.
- Both operands are scalars with numeric or **bool** type.
- Both operands are vectors with numeric or **bool** type, where the two vectors are of the same size, which is less than or equal to four.

If the first operand is a packed vector of **bool**, then the conditional selection is performed on an elementwise basis. Both the second and third operands must be numeric vectors of the same size as the first operand.

Unlike C, side effects in the expressions in the second and third operands are always executed, regardless of the condition.

Miscellaneous Operators

(typecast) ,

Cg supports C's typecast and comma operators.

Reserved Words

The following are the reserved words in Cg:

asm*	**asm_fragment**	**auto**
bool	**break**	**case**
catch	**char**	**class**
column major	**compile**	**const**
const_cast	**continue**	**decl***
default	**delete**	**discard**
do	**double**	**dword***
dynamic_cast	**else**	**emit**
enum	**explicit**	**extern**
false	**fixed**	**float***
for	**friend**	**get**
goto	**half**	**if**
in	**inline**	**inout**
int	**interface**	**long**
matrix*	**mutable**	**namespace**
new	**operator**	**out**
packed	**pass***	**pixelfragment***
pixelshader*	**private**	**protected**
public	**register**	**reinterpret_cast**
return	**row major**	**sampler**
sampler_state	**sampler1D**	**sampler2D**
sampler3D	**samplerCUBE**	**shared**
short	**signed**	**sizeof**
static	**static_cast**	**string***
struct	**switch**	**technique***
template	**texture***	**texture1D**
texture2D	**texture3D**	**textureCUBE**
textureRECT	**this**	**throw**
true	**try**	**typedef**
typeid	**typename**	**uniform**
union	**unsigned**	**using**
vector*	**vertexfragment***	**vertexshader***
virtual	**void**	**volatile**
while	***__identifier*** (two underscores before identifier)	

Cg Standard Library Functions

Cg provides a set of built-in functions and predefined structures with binding semantics to simplify GPU programming. These functions are discussed in "Cg Standard Library Functions" on page 19.

Vertex Program Profiles

A few features of the Cg language that are specific to vertex program profiles are required to be implemented in the same manner for all vertex program profiles.

Mandatory Computation of Position Output

Vertex program profiles may (and typically do) require that the program compute a position output. This homogeneous clip-space position is used by the hardware rasterizer and must be stored in a program output with an output binding semantic of **`POSITION`** (or **`HPOS`** for backward compatibility).

Position Invariance

In many graphics APIs, the user can choose between two different approaches to specifying per-vertex computations: use a built-in configurable *fixed-function* pipeline or specify a user-written vertex program. If the user wishes to mix these two approaches, it is sometimes desirable to guarantee that the position computed by the first approach is bit-identical to the position computed by the second approach. This *position invariance* is particularly important for multipass rendering.

Support for position invariance is optional in Cg vertex profiles, but for those vertex profiles that support it, the following rules apply:

- Position invariance with respect to the fixed function pipeline is guaranteed if two conditions are met:
  - The vertex program is compiled using a compiler option indicating position invariance (**`-posinv`**, for example).
  - The vertex program computes position as follows:

    `OUT_POSITION` **`= mul(`*`MVP, IN_POSITION`***`)`**

    where

    `OUT_POSITION` is a variable (or structure element) of type **`float4`** with an output binding semantic of **`POSITION`** or **`HPOS`**.

    `IN_POSITION` is a variable (or structure element) of type **`float4`** with an input binding semantic of **`POSITION`**.

    `MVP` is a uniform variable (or structure element) of type **`float4x4`** with an input binding semantic that causes it to track the fixed-function modelview-projection matrix. (The name of this binding semantic is currently profile-specific—for OpenGL profiles, the semantic **`_GL_MVP`** is recommended).

- If the first condition is met but not the second, the compiler is encouraged to issue a warning.
- Implementations may choose to recognize more general versions of the second condition (such as the variables being copy propagated from the original inputs and outputs), but this additional generality is not required.

Binding Semantics for Outputs

As shown in Table 8, there are two output binding semantics for vertex program profiles:

Table 8 Vertex Output Binding Semantics

Name	Meaning	Type	Default Value
`POSITION`	Homogeneous clip-space position; fed to rasterizer.	`float4`	Undefined
`PSIZE`	Point size	`float`	Undefined

Profiles may define additional output binding semantics with specific behaviors, and these definitions are expected to be consistent across commonly used profiles.

Fragment Program Profiles

A few features of the Cg language that are specific to fragment program profiles are required to be implemented in the same manner for all fragment program profiles.

Binding Semantics for Outputs

As shown in Table 9, there are three output binding semantics for fragment program profiles. Profiles may define additional output binding semantics with specific behaviors, and these definitions are expected to be consistent across commonly used profiles.

Table 9 Fragment Output Binding Semantics

Name	Meaning	Type	Default Value
`COLOR`	RGBA output color	`float4`	Undefined

Table 9 Fragment Output Binding Semantics (continued)

COLOR0	Same as **COLOR**	—	—
DEPTH	Fragment depth value (in range [0,1])	**float**	Interpolated depth from rasterizer (in range [0,1])

If a program desires an output color alpha of 1.0, it should explicitly write a value of 1.0 to the **W** component of the **COLOR** output. The language does *not* define a default value for this output.

Note: If the target hardware uses a default value for this output, the compiler may choose to optimize away an explicit write specified by the user if it matches the default hardware value. Such defaults are not exposed in the language.

In contrast, the language does define a default value for the **DEPTH** output. This default value is the interpolated depth obtained from the rasterizer. Semantically, this default value is copied to the output at the beginning of the execution of the fragment program.

As discussed earlier, when a binding semantic is applied to an output, the type of the output variable is not required to match the type of the binding semantic. For example, the following is legal, although not recommended:

```
struct myfragoutput {
  float2 mycolor : COLOR; }
```

In such cases, the variable is implicitly copied (with a **typecast**) to the semantic upon program completion. If the variable's vector size is shorter than the semantic's vector size, the larger-numbered components of the semantic receive their default values, if applicable, and otherwise are undefined. In the case above, the **R** and **G** components of the output color are obtained from ***mycolor***, while the **B** and **A** components of the color are undefined.

Appendix B
Language Profiles

This appendix describes the language capabilities that are available in each of the following profiles supported by the Cg compiler:

- DirectX Vertex Shader 2.x Profiles (`vs_2_0`, `vs_2_x`)
- DirectX Pixel Shader 2.x Profiles (`ps_2_0`, `ps_2_x`)
- OpenGL ARB Vertex Program Profile (`arbvp1`)
- OpenGL ARB Fragment Program Profile (`arbfp1`)
- OpenGL NV_vertex_program 2.0 Profile (`vp30`)
- OpenGL NV_fragment_program Profile (`fp30`)
- DirectX Vertex Shader 1.1 Profile (`vs_1_1`)
- DirectX Pixel Shader 1.x Profiles (`ps_1_1, ps_1_2, ps_1_3`)
- OpenGL NV_vertex_program 1.0 Profile (`vp20`)
- OpenGL NV_texture_shader and NV_register_combiners Profile (`fp20`)

In each case, the capabilities are a subset of the full capabilities described by the Cg language specification in "Cg Language Specification" on page 165.

DirectX Vertex Shader 2.x Profiles (`vs_2_0`, `vs_2_x`)

The DirectX Vertex Shader 2.0 profiles are used to compile Cg source code to DirectX 9 VS 2.0 vertex shaders[1] and DirectX 9 VS 2.0 Extended vertex shaders.

- **Profile Name**:
  `vs_2_0` (for DirectX 9 VS 2.0 vertex shaders)
  `vs_2_x` (for DirectX 9 VS 2.0 extended vertex shaders)
- **How To Invoke**:
  Use the compiler option `-profile vs_2_0` or `-profile vs_2_x`

This section describes how using the `vs_2_0` and `vs_2_x` profiles affects the Cg source code that the developer writes.

Overview

The `vs_2_0` profile limits Cg to match the capabilities of DirectX VS 2.0 vertex shaders. The `vs_2_x` profile is the same as the `vs_2_0` profile but allows extended features such as dynamic flow control (branching).

Memory

DirectX 9 vertex shaders have a limited amount of memory for instructions and data.

Program Instruction Limit

DirectX 9 vertex shaders are limited to 256 instructions. If the compiler needs to produce more than 256 instructions to compile a program, it reports an error.

Vector Register Limit

Likewise, there are limited numbers of registers to hold program parameters and temporary results. Specifically, there are 256 read-only vector registers and 12-32 read/write vector registers. If the compiler needs more registers to compile a program than are available, it generates an error.

1. To understand the capabilities of DirectX VS 2.0 Vertex Shaders and the code produced by the compiler, refer to the Vertex Shader Reference in the DirectX 9 SDK documentation.

Statements and Operators

If the **vs_2_0** profile is used, then **if**, **while**, **do**, and **for** statements are allowed only if the loops they define can be unrolled because here is no dynamic branching in unextended VS 2.0 shaders.

If the **vs_2_x** profile is used, then **if**, **while**, and **do** statements are fully supported as long as the **DynamicFlowControlDepth** option is not 0.

Comparison operators are allowed (**>**, **<**, **>=**, **<=**, **==**, **!=**) and Boolean operators (**||**, **&&**, **?:**) are allowed. However, the logic operators (**&**, **|**, **^**, **~**) are not.

Data Types

The profiles implement data types as follows:

- **float** data types are implemented as IEEE 32-bit single precision.
- **half** and **double** data types are treated as **float**.
- **int** data type is supported using floating point operations, which adds extra instructions for proper truncation for divides, modulos and casts from floating point types.
- **fixed** or **sampler*** data types are not supported, but the profiles do provide the minimal "partial support" that is required for these data types by the core language specification—that is, it is legal to declare variables using these types, as long as no operations are performed on the variables.

Using Arrays

Variable indexing of arrays is allowed as long as the array is a uniform constant. For compatibility reasons arrays indexed with variable expressions need not be declared const just uniform. However, writing to an array that is later indexed with a variable expression yields unpredictable results.

Array data is not packed because vertex program indexing does not permit it. Each element of the array takes a single 4-float program parameter register. For example, **float arr[10]**, **float2 arr[10]**, **float3 arr[10]**, and **float4 arr[10]** all consume 10 program parameter registers.

It is more efficient to access an array of vectors than an array of matrices. Accessing a matrix requires a floor calculation, followed by a multiply by a constant to compute the register index. Because vectors (and scalars) take one register, neither the floor nor the multiply is needed. It is faster to do matrix skinning using arrays of vectors with a premultiplied index than using arrays of matrices.

Bindings

Binding Semantics for Uniform Data

Table 10 summarizes the valid binding semantics for uniform parameters in the **vs_2_0** and **vs_2_X** profiles.

Table 10 **vs_2_0/vs_2_x** Uniform Input Binding Semantics

Binding Semantics	Corresponding Data
register(c0)–register(c255) **C0–C255**	Constant register [0..95]. The aliases c0-c95 (lowercase) are also accepted. If used with a variable that requires more than one constant register (for example, a matrix), the semantic specifies the first register that is used.

Binding Semantics for Varying Input/Output Data

Only the binding semantic names need be given for these profiles. The vertex parameter input registers are allocated dynamically. All the semantic names, except **POSITION**, can have a number from 0 to 15 after them.

Table 11 **vs_2_0/vs_2_x** Varying Input Binding Semanitics

Binding Semantics Name
POSITION
BLENDWEIGHT
NORMAL
COLOR
TESSFACTOR
PSIZE
BLENDINDICES
TEXCOORD
TANGENT
BINORMAL

Table 12 summarizes the valid binding semantics for varying output parameters in the **vs_2_0** and **vs_2__x** profiles.

These map to output registers in DirectX 9 vertex shaders.

Table 12 **vs_2_0/vs_2_x** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION	Output position: **oPos**
PSIZE	Output point size: **oPts**
FOG	Output fog value: **oFog**
COLOR0-COLOR1	Output color values: **oD0, oD1**
TEXCOORD0–TEXCOORD7	Output texture coordinates: **oT0–oT7**

Options

The **vs_2_x** profile allows the following profile specific options:

DynamicFlowControlDepth <*n*> (where ***n*** is 0 or 24, default 24)

NumTemps <*n*> (where ***n*** is >= 12 and <= 32, default 16)

Predication (default true)

DirectX Pixel Shader 2.x Profiles (`ps_2_0`, `ps_2_x`)

The DirectX Pixel Shader 2.0 Profiles are used to compile Cg source code to DirectX 9 PS 2.0 pixel shaders[2] and DirectX 9 PS 2.0 extended pixel shaders.

- **Profile name**:
  `ps_2_0` (for DirectX 9 PS 2.0 pixel shaders)
  `ps_2_x` (for DirectX 9 PS 2.0 extended pixel shaders)
- **How to invoke**:
  Use the compiler option `-profile ps_2_0` or `-profile ps_2_x`

The `ps_2_0` profile limits Cg to match the capabilities of DirectX PS 2.0 pixel shaders. The `ps_2_x` profile is the same as the `ps_2_0` profile but allows extended features such as arbitrary swizzles, larger limit on number of instructions, no limit on texture instructions, no limit on texture dependent reads, and support for predication.

This section describes the capabilities and restrictions of Cg when using these profiles.

Memory

Program Instruction Limit

DirectX 9 Pixel shaders have a limit on the number of instructions in a pixel shader.

- PS 2.0 (`ps_2_0`) pixel shaders are limited to 32 texture instructions and 64 arithmetic instructions.
- Extended PS 2 (`ps_2_x`) shaders have a limit of maximum number of total instructions between 96 to 1024 instructions.
  There is no separate texture instruction limit on extended pixel shaders.

If the compiler needs to produce more than the maximum allowed number of instructions to compile a program, it reports an error.

Vector Register Limit

Likewise, there are limited numbers of registers to hold program parameters and temporary results. Specifically, there are 32 read-only vector registers and 12-32 read/write vector registers. If the compiler needs more registers to compile a program than are available, it generates an error.

2. To understand the capabilities of DirectX PS 2.0 Pixel Shaders and the code produced by the compiler, refer to the Pixel Shader Reference in the DirectX 9 SDK documentation.

Language Constructs and Support

Data Types

This profile implements data types as follows:

- **`float`** data type is implemented as IEEE 32-bit single precision.
- **`half`**, **`fixed`**, and **`double`** data types are treated as float.
  `half` data types can be used to specify partial precision hint for pixel shader instructions.
- **`int`** data type is supported using floating point operations.
- **`sampler*`** types are supported to specify sampler objects used for texture fetches.

Statements and Operators

With the **`ps_2_0`** profiles **`while`**, **`do`**, and **`for`** statements are allowed only if the loops they define can be unrolled because there is no dynamic branching in PS 2.0 shaders. In current Cg implementation, extended **`ps_2_x`** shaders also have the same limitation.

Comparison operators are allowed (**`>`**, **`<`**, **`>=`**, **`<=`**, **`==`**, **`!=`**) and Boolean operators (**`||`**, **`&&`**, **`?:`**) are allowed. However, the logic operators (**`&`**, **`|`**, **`^`**, **`~`**) are not.

Using Arrays and Structures

Variable indexing of arrays is not allowed. Array and structure data is not packed.

Bindings

Binding Semantics for Uniform Data

Table 13 summarizes the valid binding semantics for uniform parameters in the **ps_2_0** and **ps_2_X** profiles

Table 13 **ps_2_0/ps_2_x** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
register(s0)–register(s15) **TEXUNIT0-TEXUNIT15**	Texunit unit ***N***, where ***N*** is in range [0..15] May only be used with uniform inputs with **sampler*** types.
register(c0)-register(c31) **C0–C31**	Constant register ***N***, where ***N*** is in range [0..31] May only be used with uniform inputs.

Binding Semantics for Varying Input/Output Data

Table 14 summarizes the valid binding semantics for varying input parameters in the **ps_2_0** and **ps_2_x** profiles.

Table 14 **ps_2_0/ps_2_x** Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data (type)
COLOR0	Input color0 (**float4**)
COLOR1	Input color1 (**float4**)
TEXCOORD0-TEXCOORD7	Input texture coordinates (**float4**)

Table 15 summarizes the valid binding semantics for varying output parameters in the **ps_2_0** and **ps_2_x** profiles.

Table 15 **ps_2_0/ps_2_x** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0	Output color (**float4**)
DEPTH	Output depth (**float**)

Options

The **ps_2_x** profile allows the following profile specific options:

NumTemps=<*n*> (where ***n*** is >= 12 and <= 32; default 32)

NumInstructionSlots=<*n*> (where ***n*** is >= 96 and <= 1024; default 1024)

Predication=<*b*> (where ***b***=0 or 1; default is 1)

ArbitrarySwizzle=<*b*> (where ***b***=0 or 1; default is 1)

GradientInstructions=<*b*> (where ***b***=0 or 1; default is 1)

NoDependentReadLimit=<*b*> (where ***b***=0 or 1; default is 1)

NoTexInstructionLimit=<*b*> (where ***b***=0 or 1; default is 1)

Limitations in this Implementation

Currently, this profile implementation has the following limitations:

- Dynamic flow control is not supported in extended pixel shaders.
- Multiple color outputs are not supported in pixel shaders. Only **Color0** is supported.

OpenGL ARB Vertex Program Profile (`arbvp1`)

The OpenGL ARB Vertex Program Profile is used to compile Cg source code to vertex programs compatible with version 1.0 of the **`GL_ARB_vertex_program`** extension.

- **Profile Name**:
  `arbvp1`
- **How To Invoke**:
  Use the compiler option **`-profile arbvp1`**

This section describes the capabilities and restrictions of Cg when using the **`arbvp1`** profile.

Overview

- The **`arbvp1`** profile is similar to the **`vp20`** profile except for the format of its output and its capability of accessing OpenGL state easily.
- **`ARB_vertex_program`** has the same capabilities as **`NV_vertex_program`** and DirectX 8 vertex shaders, so the limitations that this profile places on the Cg source code written by the programmer is the same as the **`NV_vertex_program`**[3] profile.

Accessing OpenGL State

The **`arbvp1`** profile allows Cg programs to refer to the OpenGL state directly, unlike the **`vp20`** profile. However, if you want to write Cg programs that are compatible with **`vp20`** and **`dx8vs`** profiles, you should use the alternate mechanism of setting uniform variables with the necessary state using the Cg run time. The compiler relies on the feature of ARB vertex assembly programs that enables parts of the OpenGL state to be written automatically to program parameter registers as the state changes. The OpenGL driver handles this state-tracking feature. A special variable called **`glstate`**, defined as a structure, can be used to refer to every part of the OpenGL state that ARB vertex programs can reference. Following this paragraph are three lists of the **`glstate`** fields that can be accessed. The array indexes are shown as **`0`**, but an array can be accessed using any positive integer that is less than the limit of the array. For example, to access the diffuse component of the second light use **`glstate.light[1].diffuse`**, assuming that **`GL_MAX_LIGHTS`** is at least **`2`**.

3. See "DirectX Vertex Shader 1.1 Profile (vs_1_1)" on page 222 for a full explanation of the data types, statements, and operators supported by this profile.

`float4x4 glstate` Fields

These are the **`glstate`** fields of type **`float4x4`** that can be accessed:

```
glstate.matrix.modelview[0]          glstate.matrix.projection
glstate.matrix.mvp                   glstate.matrix.texture[0]
glstate.matrix.palette[0]            glstate.matrix.program[0]
glstate.matrix.inverse.modelview[0]  glstate.matrix.inverse.projection
glstate.matrix.inverse.mvp           glstate.matrix.inverse.texture[0]
glstate.matrix.inverse.palette[0]    glstate.matrix.inverse.program[0]
glstate.matrix.transpose.modelview[0]      glstate.matrix.transpose.projection
glstate.matrix.transpose.mvp         glstate.matrix.transpose.texture[0]
glstate.matrix.transpose.palette[0]  glstate.matrix.transpose.program[0]
glstate.matrix.invtrans.modelview[0] glstate.matrix.invtrans.projection
glstate.matrix.invtrans.mvp          glstate.matrix.invtrans.texture[0]
glstate.matrix.invtrans.palette[0]   glstate.matrix.invtrans.program[0]
```

`float4 glstate` Fields

These are the **`glstate`** fields of type **`float4`** that can be accessed:

```
glstate.material.ambient             glstate.material.diffuse
glstate.material.specular            glstate.material.emission
glstate.material.shininess           glstate.material.front.ambient
glstate.material.front.diffuse       glstate.material.front.specular
glstate.material.front.emission      glstate.material.front.shininess
glstate.material.back.ambient        glstate.material.back.diffuse
glstate.material.back.specular       glstate.material.back.emission
glstate.material.back.shininess      glstate.light[0].ambient
glstate.light[0].diffuse             glstate.light[0].specular
glstate.light[0].position            glstate.light[0].attenuation
glstate.light[0].spot.direction      glstate.light[0].half
glstate.lightmodel.ambient           glstate.lightmodel.scenecolor
glstate.lightmodel.front.scenecolor glstate.lightmodel.back.scenecolor
glstate.lightprod[0].ambient         glstate.lightprod[0].diffuse
glstate.lightprod[0].specular        glstate.lightprod[0].front.ambient
glstate.lightprod[0].front.diffuse   glstate.lightprod[0].front.specular
glstate.lightprod[0].back.ambient    glstate.lightprod[0].back.diffuse
glstate.lightprod[0].back.specular   glstate.texgen[0].eye.s
glstate.texgen[0].eye.t              glstate.texgen[0].eye.r
glstate.texgen[0].eye.q              glstate.texgen[0].object.s
glstate.texgen[0].object.t           glstate.texgen[0].object.r
glstate.texgen[0].object.q           glstate.fog.color
glstate.fog.params                   glstate.clip[0].plane
```

`float glstate` Fields

These are the **`glstate`** fields of type **`float`** that can be accessed:

```
glstate.point.size
glstate.point.attenuation
```

Position Invariance

- The **`arbvp1`** profile supports position invariance, as described in the core language specification.
- The modelview-projection matrix is not specified using a binding semantic of **`_GL_MVP`**.

Data Types

This profile implements data types as follows:

- **`float`** data type is implemented as defined in the **`ARB_vertex_program`** specification.
- **`half`** data type is implemented as float.
- **`fixed`** or **`sampler*`** data types are not supported, but the profile does provide the minimal 'partial support' that is required for these data types by the core language specification—that is, it is legal to declare variables using these types as long as no operations are performed on the variables.

Compatibility with the `vp20` Vertex Program Profile

Programs that work with the **`vp20`** profile are compatible with the **`arbvp1`** profile as long as they use the Cg run time to manage all uniform parameters, including OpenGL state. That is, **`arbvp1`** and **`vp20`** profiles can be used interchangeably without changing the Cg source code or the application program except for specifying a different profile. However, if any of the **`glProgramParameterxxNV()`** routines are used the application program needs to be changed to use the corresponding ARB functions.

Since there is no ARB function corresponding to **`glTrackMatrixNV()`**, an application using **`glTrackMatrixNV()`** and the **`arbvp1`** profile needs to be modified. One solution is to change the Cg source code to refer to the matrix using the **`glstate`** structure so that the matrix is automatically tracked by the OpenGL driver as part of its **`GL_ARB_vertex`** support. Another solution is for the application to use the Cg run-time routine **`cgGLSetStateMatrixParameter()`** to load the appropriate matrix or matrices when necessary.

Another potential incompatibility between the **arbvp1** and **vp20** profiles is the way that input varying semantics are handled. In the **vp20** profile, semantic names such as **POSITION** and **ATTR0** are aliases of each other the same way **NV_vertex_program** aliases Vertex and Attribute 0 (see Table 39, "vp20 Varying Input Binding Semantics," on page 242). In the **arbvp1** profile, the semantic names are not aliased because **ARB_vertex_program** allows the conventional attributes (such as vertex position) to be separate from the generic attributes (such as Attribute 0). For this reason it is important to follow the conventions given in Table 17, "arbvp1 Varying Input Binding Semantics," on page 210 so that **arbvp1** programs work for all implementations of **ARB_vertex_program**. The **arbvp1** conventions are compatible with the **vp20** and **vp30** profiles.

Loading Constants

Applications that do not use the Cg run time are no longer required to load constant values into program parameters registers as indicated by the **#const** expressions in the Cg compiler output. The compiler produces output that causes the OpenGL driver to load them. However, uniform variables that have a default definition still require constant values to be loaded into the appropriate program parameter registers, as ARB vertex programs do not support this feature. Application programs either have to use the Cg run time, parse, and handle the **#default** commands, or have to avoid initializing uniform variables in the Cg source code.

Bindings

Binding Semantics for Uniform Data

Table 16 summarizes the valid binding semantics for uniform parameters in the **arbvp1** profile. .

Table 16 **arbvp1** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
register(c0)–register(c255) **C0–C255**	Local parameter with index **n**, **n** = [0..255]. The aliases c0–c255 (lowercase) are also accepted. If used with a variable that requires more than one constant register (for example, a matrix), the semantic specifies the first local parameter that is used.

Binding Semantics for Varying Input/Output Data

Table 17 summarizes the valid binding semantics for uniform parameters in the **arbvp1** profile.

The set of binding semantics for varying input data to **arbvp1** consists of **POSITION**, **BLENDWEIGHT**, **NORMAL**, **COLOR0**, **COLOR1**, **TESSFACTOR**, **PSIZE**, **BLENDINDICES**, and **TEXCOORD0–TEXCOORD7**. One can also use **TANGENT** and **BINORMAL** instead of **TEXCOORD6** and **TEXCOORD7**. Additionally, a set of binding semantics of **ATTR0–ATTR15** can be used. The mapping of these semantics to corresponding setting command is listed in the table.

Table 17 **arbvp1** Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION	Input Vertex, through **Vertex** command
BLENDWEIGHT	Input vertex weight through **WeightARB**, **VertexWeightEXT** command
NORMAL	Input normal through **Normal** command
COLOR0, DIFFUSE	Input primary color through **Color** command
COLOR1, SPECULAR	Input secondary color through **SecondaryColorEXT** command
FOGCOORD	Input fog coordinate through **FogCoordEXT** command
TEXCOORD0-TEXCOORD7	Input texture coordinates (**texcoord0-texcoord7**) through **MultiTexCoord** command
ATTR0-ATTR15	Generic Attribute 0-15 through **VertexAttrib** command
PSIZE, ATTR6	Generic Attribute 6

Table 18 summarizes the valid binding semantics for varying output parameters in the **arbvp1** profile.

These binding semantics map to **ARB_vertex_program** output registers. The two sets act as aliases to each other.

Table 18 **arbvp1** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION, HPOS	Output position
PSIZE, PSIZ	Output point size
FOG, FOGC	Output fog coordinate
COLOR0, COL0	Output primary color
COLOR1, COL1	Output secondary color
BCOL0	Output backface primary color
BCOL1	Output backface secondary color
TEXCOORD0-TEXCOORD7, TEX0-TEX7	Output texture coordinates

Note: The application must call **glEnable(GL_COLOR_SUM_ARB)** in order to enable **COLOR1** output when using the **arbvp1** profile.

The profile also allows **WPOS** to be present as binding semantics on a member of a structure of a varying output data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have the same structure specify the varying output of an **arbvp1** profile program and the varying input of an **fp30** profile program.

OpenGL ARB Fragment Program Profile (`arbfp1`)

The OpenGL ARB Fragment Program Profile is used to compile Cg source code to fragment programs compatible with version 1.0 of the **`GL_ARB_fragment_program`** OpenGL extension.[4]

- **Profile name**: **`arbfp1`**
- **How to invoke**: Use the compiler option **`-profile arbfp1`**

The **`arbfp1`** profile limits Cg to match the capabilities of OpenGL ARB fragment programs. This section describes the capabilities and restrictions of Cg when using the **`arbfp1`** profile.

Memory

Program Instruction Limits

OpenGL ARB fragment programs have a limit on number of instructions in an ARB fragment program.

ARB fragment programs are limited to number of instructions that can be queried from underlying OpenGL implementation using **`MAX_PROGRAM_INSTRUCTIONS_ARB`** with a minimum value of 72. There are limits on number of texture instructions (minimum limit of 24) and arithmetic instructions (minimum limit of 48) that can be queried from OpenGL implementation.

If the compiler needs to produce more than maximum allowed instructions to compile a program, it reports an error.

Vector Register Limits

Likewise, there are limited numbers of registers that can be queried from OpenGL implementation to hold local program parameters (minimum limit of 24) and temporary results (minimum limit of 16).

If the compiler needs more temporaries or local parameters to compile a program than are available, it generates an error.

4. To understand the capabilities of OpenGL ARB fragment programs and the code produced by the compiler, refer to the ARB fragment program extension in the OpenGL Extensions documentation.

Language Constructs and Support

Data Types

This profile implements data types as follows:

- **`float`** data type is implemented as IEEE 32-bit single precision.
- **`half`**, **`fixed`**, and **`double`** data types are treated as float.
- **`int`** data type is supported using floating point operations.
- **`sampler*`** types are supported to specify sampler objects used for texture fetches.

Statements and Operators

With the ARB fragment program profiles **`while`**, **`do`**, and **`for`** statements are allowed only if the loops they define can be unrolled because there is no dynamic branching in ARB fragment program 1.

Comparison operators are allowed (**`>`**, **`<`**, **`>=`**, **`<=`**, **`==`**, **`!=`**) and Boolean operators (**`||`**, **`&&`**, **`?:`**) are allowed. However, the logic operators (**`&`**, **`|`**, **`^`**, **`~`**) are not.

Using Arrays and Structures

Variable indexing of arrays is not allowed. Array and structure data is not packed.

Bindings

Binding Semantics for Uniform Data

Table 19 summarizes the valid binding semantics for uniform parameters in the **`arbfp1`** profile.

Table 19 **`arbfp1`** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
`register(s0)—register(s15)` **`TEXUNIT0-TEXUNIT15`**	Texunit image unit ***`N`***, where ***`N`*** is in range [0..15] May only be used with uniform inputs with **`sampler*`** types.
`register(c0)-register(c31)` **`C0–C31`**	Local Parameter ***`N`***, where ***`N`*** is in range [0..31] May only be used with uniform inputs.

Binding Semantics for Varying Input/Output Data

Table 20 summarizes the valid binding semantics for varying input parameters in the **arbfp1** profile

Table 20 **arbfp1** Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data (type)
COLOR0	Input color0 (float4)
COLOR1	Input color1 (float4)
TEXCOORD0-TEXCOORD7	Input texture coordinates (float4)

Table 21 summarizes the valid binding semantics for varying output parameters in the **arbfp1** profile.

Table 21 **arbfp1** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0	Output color (float4)
DEPTH	Output depth (float)

Options

The ARB fragment program profile allows the following profile specific options:

NumTemps=<*n*> (where ***n*** is >= 16 and <= 32, default 32)

NumInstructionSlots=<*n*> (where ***n*** is >= 72 and <= 1024, default 1024)

NoDependentReadLimit=<*b*> (where ***b***=0 or 1. default is 1)

NumTexInstructionSlots=<*n*> (where ***n***>=24)

Limitations in the Implementation

Currently, this profile implementation has following limitations:

- OpenGL ARB fragment program profile is still in developmental beta stage as the extension and its support is not widely available.
- OpenGL state access in ARB fragment programs is not yet implemented.

OpenGL NV_vertex_program 2.0 Profile (**vp30**)

The **vp30** Vertex Program profile is used to compile Cg source code to vertex programs for use by the **NV_vertex_program2** OpenGL extension.

- **Profile name**: **vp30**
- **How to invoke**: Use the compiler option **-profile vp30**

The **vp30** profile limits Cg to match the capabilities of the **NV_vertex_program2** extension. This section describes the capabilities and restrictions of Cg when using the **vp30** profile.

Position Invariance

The **vp30** profile supports position invariance, as described in the core language specification.

- The modelview-projection matrix must be specified using a binding semantic of **_GL_MVP**. Unlike the **vp20** and **arbvp1** profiles, this profile causes the compiler to emit the instructions for transforming the position using the modelview-projection matrix.
- The assembly code position invariant option is not used because the hardware guarantees that the position calculation is invariant compared to the fixed pipeline calculation.

Language Constructs

Data Types

This profile implements data types as follows:

- **float** data type is implemented as IEEE 32-bit single precision.
- **half** data type is implemented as **float**.
- **int** data type is supported using floating point operations, which adds extra instructions for proper truncation for divides, modulos, and casts from floating point types.
- **fixed** or **sampler*** data types are not supported, but the profile does provide the minimal "partial support" that is required for these data types by the core language specification—that is, it is legal to declare variables using these types, as long as no operations are performed on the variables.

Statements and Operators

This profile is a superset of the **vp20** profile. Any program that compiles for the **vp20** profile should also compile for the **vp30** profile, although the converse is not true.

The additional capabilities of the **vp30** profile, beyond those of **vp20** are:

- **for**, **while**, and **do** loops are supported without requiring loop unrolling
- Full support for **if/else** allowing non-constant conditional expressions

Bindings

Binding Semantics for Uniform Data

Table 22 summarizes the valid binding semantics for uniform parameters in the **vp30** profile.

Table 22 **vp30** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
register(c0)–register(c255) **C0–C255**	Constant register [0..255]. The aliases c0–c255 (lowercase) are also accepted. If used with a variable that requires more than one constant register (for example, a matrix), the semantic specifies the first register that is used.

Binding Semantics for Varying Input/Output Data

Table 23 summarizes the valid binding semantics for varying input parameters in the **vp30** profile.

One can also use **TANGENT** and **BINORMAL** instead of **TEXCOORD6** and **TEXCOORD7**. These binding semantics map to **NV_vertex_program2** input attribute parameters. The two sets act as aliases to each other.

Table 23 **vp30** Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION, ATTR0	Input Vertex, Generic Attribute 0
BLENDWEIGHT, ATTR1	Input vertex weight, Generic Attribute 1
NORMAL, ATTR2	Input normal, Generic Attribute 2
COLOR0, DIFFUSE, ATTR3	Input primary color, Generic Attribute 3
COLOR1, SPECULAR, ATTR4	Input secondary color, Generic Attribute 4
TESSFACTOR, FOGCOORD, ATTR5	Input fog coordinate, Generic Attribute 5
PSIZE, ATTR6	Input point size, Generic Attribute 6
BLENDINDICES, ATTR7	Generic Attribute 7
TEXCOORD0-TEXCOORD7, ATTR8-ATTR15	Input texture coordinates (**texcoord0-texcoord7**), Generic Attributes 8–15
TANGENT, ATTR14	Generic Attribute 14
BINORMAL, ATTR15	Generic Attribute 15

Table 24 summarizes the valid binding semantics for varying output parameters in the **vp30** profile.

These binding semantics map to **NV_vertex_program2** output registers. The two sets act as aliases to each other.

Table 24 **vp30** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION, HPOS	Output position
PSIZE, PSIZ	Output point size

Table 24 **vp30** Varying Output Binding Semantics (continued)

Binding Semantics Name	Corresponding Data
FOG, FOGC	Output fog coordinate
COLOR0, COL0	Output primary color
COLOR1, COL1	Output secondary color
BCOL0	Output backface primary color
BCOL1	Output backface secondary color
TEXCOORD0-TEXCOORD7, TEX0-TEX7	Output texture coordinates
CLP0-CL5	Output Clip distances

The profile allows **WPOS** to be present as binding semantics on a member of a structure of a varying output data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have same structure specify the varying output of a **vp30** profile program and the varying input of an **fp30** profile program.

OpenGL NV_fragment_program Profile (`fp30`)

The **`fp30`** Fragment Program Profile is used to compile Cg source code to fragment programs for use by the **`NV_fragment_program`** OpenGL extension.

- **Profile Name**: **`fp30`**
- **How To Invoke**: Use the compiler option **`-profile fp30`**

This section describes the capabilities and restrictions of Cg when using the **`fp30`** profile.

Language Constructs and Support

Data Types

- **`fixed`** type (s1.10 fixed point) is supported
- **`half`** type (s10e5 floating-point) is supported

It is recommended that you use **`fixed`**, **`half`**, and **`float`** in that order for maximum performance. Reversing this order provides maximum precision. You are encouraged to use the fastest type that meets your needs for precision.

Statements and Operators

- Full support for **`if/else`**
- No **`for`** and **`while`** loops, unless they can be unrolled by the compiler
- Support for flexible texture mapping
- Support for screen-space derivative functions
- No support for variable indexing of arrays.

Bindings

Binding Semantics for Uniform Data

Table 25 summarizes the valid binding semantics for uniform parameters in the **fp30** profile.

Table 25 **fp30** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
register(s0)-register(s15) **TEXUNIT0-TEXUNIT15**	Texunit ***N***, where ***N*** is in the range [0..15]. May be used only with uniform inputs with **sampler*** types.
register(c0)-register(c31) **C0-C31**	Constant register ***N***, where ***N*** is in range [0..15] May only be used with uniform inputs.

Binding Semantics for Varying Input/Ouput Data

Table 26 summarizes the valid binding semantics for varying input parameters in the **fp30** profile.

These binding semantics map to **NV_fragment_program** input registers. The two sets act as aliases to each other. The profile also allows **POSITION**, **FOG**, **PSIZE**, **HPOS**, **FOGC**, **PSIZ**, **BCOL0**, **BCOL1**, and **CLP0–CLP5** to be present as binding semantics on a member of a structure of a varying input data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have the same structure specify the varying output of a **vp30** profile program and the varying input of an **fp30** profile program. .

Table 26 **fp30** Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data (type)
COLOR0, COL0	Input color0 (**float4**)
COLOR1, COL1	Input color1 (**float4**)
TEXCOORD0-TEXCOORD7, **TEX0-TEX7**	Input texture coordinates (**float4**)
WPOS	Window Position Coordinates (**float4**)

Table 27 summarizes the valid binding semantics for varying output parameters in the **fp20** profile. .

Table 27 **fp30** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0, COL	Output color (**float4**)
DEPTH, DEPR	Output depth (**float**)

DirectX Vertex Shader 1.1 Profile (`vs_1_1`)

The DirectX Vertex Shader 1.1 profile is used to compile Cg source code to DirectX 8.1 Vertex Shaders and DirectX 9 VS 1.1 shaders[5].

- **Profile Name:** `vs_1_1`
- **How To Invoke:** Use the compiler option `-profile vs_1_1`.

The `vs_1_1` profile limits Cg to match the capabilities of DirectX Vertex Shaders.

This section describes how using the `vs_1_1` profile affects the Cg source code that the developer writes.

Memory Restrictions

DirectX 8 vertex shaders have a limited amount of memory for instructions and data.

Program Instruction Limits

The DirectX 8 vertex shaders are limited to 128 instructions. If the compiler needs to produce more than 128 instructions to compile a program, it reports an error.

Vector Register Limits

Likewise, there are limited numbers of registers to hold program parameters and temporary results. Specifically, there are 96 read-only vector registers and 12 read/write vector registers. If the compiler needs more registers to compile a program than are available, it generates an error.

Language Constructs and Support

Data Types

This profile implements data types as follows:

- `float` data types are implemented as IEEE 32-bit single precision.
- `half` and `double` data types are treated as `float`.

5. To understand the capabilities of DirectX VS 1.1 Vertex Shaders and the code produced by the compiler, refer to the Vertex Shader Reference in the DirectX 8.1 SDK documentation.

- **int** data type is supported using floating point operations, which adds extra instructions for proper truncation for divides, modulos and casts from floating point types.
- **fixed** or **sampler*** data types are not supported, but the profile does provide the minimal "partial support" that is required for these data types by the core language specification—that is, it is legal to declare variables using these types, as long as no operations are performed on the variables.

Statements and Operators

The **if**, **while**, **do**, and **for** statements are allowed only if the loops they define can be unrolled, because there is no branching in VS 1.1 shaders.

There are no subroutine calls either, so all functions are inlined. Comparison operators are allowed (**>**, **<**, **>=**, **<=**, **==**, **!=**) and Boolean operators (**||**, **&&**, **?:**) are allowed. However, the logic operators (**&**, **|**, **^**, **~**) are not allowed.

Using Arrays

Variable indexing of arrays is allowed as long as the array is a uniform constant. For compatibility reasons arrays indexed with variable expressions need not be declared const just uniform. However, writing to an array that is later indexed with a variable expression yields unpredictable results.

Array data is not packed because vertex program indexing does not permit it. Each element of the array takes a single 4-float program parameter register. For example, **float arr[10]**, **float2 arr[10]**, **float3 arr[10]**, and **float4 arr[10]** all consume ten program parameter registers.

It is more efficient to access an array of vectors than an array of matrices. Accessing a matrix requires a floor calculation, followed by a multiply by a constant to compute the register index. Because vectors (and scalars) take one register, neither the floor nor the multiply is needed. It is faster to do matrix skinning using arrays of vectors with a premultiplied index than using arrays of matrices.

Constants

Literal constants can be used with this profile, but it is not possible to store them in the program itself. Instead the compiler will issue, as comments, a list of program parameter registers and the constants that need to be loaded into them. The Cg run-time system will handle loading the constants, as directed by the compiler.

Note: If the Cg run-time system is not used, it is the responsibility of the programmer to make sure that the constants are loaded properly.

Bindings

Binding Semantics for Uniform Data

Table 28 summarizes the valid binding semantics for uniform parameters in the **`vs_1_1`** profile.

Table 28 **`vs_1_1`** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
`register(c0)-register(c95)` **`C0-C95`**	Constant register [0..95]. The aliases c0–c95 (lowercase) are also accepted. If used with a variable that requires more than one constant register (for example, a matrix), the semantic specifies the first register that is used.

Binding Semantics for Varying Input/Output Data

Table 29 summarizes the valid binding semantics for uniform parameters in the **`vs_1_1`** profile. These map to the input registers in DirectX 8.1 vertex shaders.

Table 29 **`vs_1_1`** Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data
`POSITION`	Vertex shader input register: **`v0`**
`BLENDWEIGHT`	Vertex shader input register: **`v1`**

Table 29 **vs_1_1** Varying Input Binding Semantics (continued)

Binding Semantics Name	Corresponding Data
BLENDINDICES	Vertex shader input register: **v2**
NORMAL	Vertex shader input register: **v3**
PSIZE	Vertex shader input register: **v4**
COLOR0, DIFFUSE	Vertex shader input register: **v5**
COLOR1, SPECULAR	Vertex shader input register: **v6**
TEXCOORD0–TEXCOORD7	Vertex shader input register: **v7–v14**
TANGENT[i]	Vertex shader input register: **v14**
BINORMAL	Vertex shader input register: **v15**

i. **TANGENT** is as alias for **TEXCOORD7**.

Table 30 summarizes the valid binding semantics for varying output parameters in the **vs_1_X** profile.These map to output registers in DirectX 8.1 vertex shaders.

Table 30 **vs_1_1** Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION	Output position: **oPos**
PSIZE	Output point size: **oPts**
FOG	Output fog value: **oFog**
COLOR0–COLOR1	Output color values: **oD0, oD1**
TEXCOORD0–TEXCOORD7	Output texture coordinates: **oT0–oT7**

Options

When using the **vs_1_1** profile under DirectX 9 it is necessary to tell the compiler to produce **dcl** statements to declare varying inputs. The option **-profileopts dcls** will cause **dcl** statements to be added to the compiler output.

DirectX Pixel Shader 1.x Profiles (`ps_1_1, ps_1_2, ps_1_3`)

The DirectX pixel shader 1_X profiles are used to compile Cg source code to DirectX PS 1.1, 1.2 or 1.3 pixel shader assembly.

- **Profile Names**:
  `ps_1_1` (for DirectX PS 1.1 pixel shaders)
  `ps_1_2` (for DirectX PS 1.2 pixel shaders)
  `ps_1_3` (for DirectX 1.2 pixel shaders)
- **How To Invoke**:
  Use the compiler option
  `-profile ps_1_1`,
  `-profile ps_1_2`, or
  `-profile ps_1_3`

The deprecated profile `dx8ps` is also available and is synonymous with `ps_1_1`.

This document describes the capabilities and restrictions of Cg when using the DirectX pixel shader 1_X profiles.

Overview

DirectX PS 1.4 is not currently supported by any Cg profile; all statements about `ps_1_X` in the remainder of this document refer only to `ps_1_1`, `ps_1_2` and `ps_1_3`.

The underlying instruction set and machine architecture limit programmability in these profiles compared to what is allowed by Cg constructs[6]. Thus, these profiles place additional restrictions on what can and cannot be done in a Cg program.

The main differences between these profiles from the Cg perspective is that additional texture addressing operations are exposed in `ps_1_2` and `ps_1_3` and the depth value output is made available (in a limited form) in `ps_1_3`.

Operations in the DirectX pixel shader 1_X profiles can be categorized as texture addressing operations and arithmetic operations. Texture addressing operations are operations which generate texture addressing instructions, arithmetic operations are operations which generate arithmetic instructions. A Cg program in one of these profiles is limited to generating a maximum of four texture addressing instructions and eight arithmetic instructions. Since these

6. For more details about the underlying instruction sets, their capabilities, and their limitations, please refer to the MSDN documentation of DirectX pixel shaders 1.1, 1.2 and 1.3.

numbersare quite small, users need to be very aware of this limitation while writing Cg code for these profiles.

There are certain simple arithmetic operations that can be applied to inputs of texture addressing operations and to inputs and outputs of arithmetic operations without generating an arithmetic instruction. From here on, these operations are referred to as input modifiers and output modifiers.

The **ps_1_X** profiles also restrict when a texture addressing operation or arithmetic operation can occur in the program. A texture addressing operation may not have any dependency on the output of an arithmetic operation unless

- the arithmetic operation is a valid input modifier for the texture addressing operation
- the arithmetic operation is part of a complex texture addressing operation (which are summarized in the section on Auxiliary Texture Functions)

Modifiers

Input and output modifiers may be used to perform simple arithmetic operations without generating an arithmetic instruction. Instead, the arithmetic operation modifies the assembly instruction or source registers to which it is applied. For example, the following Cg expression:

```
z = (x - 0.5 + y) / 2
```

could generate the following pixel shader instruction (assuming **x** is in **t0**, **y** is in **t1**, and **z** is in **r0**):

```
add_d2 r0, t0_bias, t1
```

Table 31 summarizes how different DirectX pixel shader 1_X instruction set modifiers are expressed in Cg programs. For more details on the context in which each modifier is allowed and ways in which modifiers may be combined refer to the DirectX pixel shader 1_X documentation.

Table 31 DirectX pixel shader 1_X Instruction Set Modifiers

Instruction/Register Modifier	Cg expression
instr_x2	2*x
instr_x4	4*x
instr_d2	x/2
instr_sat	saturate(x) (i.e. min(x, max(x, 1), 0))
reg_bias	x-0.5

Table 31 DirectX pixel shader 1_X Instruction Set Modifiers

Instruction/Register Modifier	Cg expression
1-**reg**	**1-x**
-**reg**	**-x**
reg_bx2	**2*(x-0.5)**

Language Constructs and Support

Data Types

In the **ps_1_X** profiles, operations occur on signed clamped floating point values in the range **MaxPixelShaderValue** to **MaxPixelShaderValue**, where **MaxPixelShaderValue** is determined by the DirectX implementation. These profiles allow all data types to be used, but all operations are carried out in the above range.

Refer to the DirectX pixel shader 1_X documentation for more details.

Statements and Operators

The DirectX pixel shader 1_X profiles support all of the Cg language constructs, with the following exceptions:

- Arbitrary swizzles are not supported (though arbitrary write masks are). Only the following swizzles are allowed
  .x/.r .y/.g .z/.b .w/.a
  .xy/.rg .xyz/.rgb .xyzw/.rgba
  .xxx/.rrr .yyy/.ggg .zzz/.bbb .www/.aaa
  .xxxx/.rrrr .yyyy/.gggg .zzzz/.bbbb .wwww/.aaaa
- Matrix swizzles are not supported.
- Boolean operators other than **<**, **<=**, > and **>=** are not supported. Furthermore, **<**, **<=**, **>** and **>=** are only supported as the condition in the **?:** operator.
- Bitwise integer operators are not supported.
- **/** is not supported unless the divisor is a non-zero constant or it is used to compute the depth output in **ps_1_3**.
- **%** is not supported.

- Ternary **?:** is supported if the boolean test expression is a compile-time boolean constant, a uniform scalar boolean or a scalar comparison to a constant value in the range [-0.5, 1.0] (for example, **a > 0.5 ? b : c**).
- **do**, **for**, and **while** loops are supported only when they can be completely unrolled.
- arrays, vectors, and matrices may be indexed only by compile-time constant values or index variables in loops that can be completely unrolled.
- The **discard** statement is not supported. The similar but less general **clip()** function is supported.
- The use of an "allocation-rule-identifier" for input and output **structs** is optional.

Standard Library Functions

Because the DirectX pixel shader 1_X profiles have limited capabilities, not all of the Cg standard library functions are supported. Table 32 presents the Cg standard library functions that are supported by these profiles. See the standard library documentation for descriptions of these functions.

Table 32 Supported Standard Library Functions

dot(floatN, floatN)
lerp(floatN, floatN, floatN)
lerp(floatN, floatN, float)
tex1D(sampler1D, float)
tex1D(sampler1D, float2)
tex1Dproj(sampler1D, float2)
tex1Dproj(sampler1D, float3)
tex2D(sampler2D, float2)
tex2D(sampler2D, float3)
tex2Dproj(sampler2D, float3)
tex2Dproj(sampler2D, float4)
texRECT(samplerRECT, float2)
texRECT(samplerRECT, float3)
texRECTproj(samplerRECT, float3)

Table 32 Supported Standard Library Functions (continued)

texRECTproj(samplerRECT, float4)
tex3D(sampler3D, float3)
tex3Dproj(sampler3D, float4)
texCUBE(samplerCUBE, float3)
texCUBEproj(samplerCUBE, float4)

Note: The non-projective texture lookup functions are actually done as projective lookups on the underlying hardware. Because of this, the **`w`** component of the texture coordinates passed to these functions from the application or vertex program must contain the value 1.

Texture coordinate parameters for projective texture lookup functions must have swizzles that match the swizzle done by the generated texture addressing instruction. While this may seem burdensome, it is intended to allow **`ps_1_X`** profile programs to behave correctly under other pixel shader profiles.

Table 33 lists the swizzles required on the texture coordinate parameter to the projective texture lookup functions.

Table 33 Required Projective Texture Lookup Swizzles

Texture Lookup Function	Texture Coordinate Swizzle
tex1Dproj	.xw/.ra
tex2Dproj	.xyw/.rga
texRECTproj	.xyw/.rga
tex3Dproj	.xyzw/.rgba
texCUBEproj	.xyz/.rgba

Bindings

Manual Assignment of Bindings

The Cg compiler can determine bindings between texture units and uniform sampler parameters/texture coordinate inputs automatically. This automatic assignment is based on the context in which uniform sampler parameters and texture coordinate inputs are used together.

To specify bindings between texture units and uniform parameters/texture coordinates to match their application, all sampler uniform parameters and texture coordinate inputs that are used in the program must have matching binding semantics—that is, **`TEXUNIT<n>`** may only be used with **`TEXCOORD<n>`**.

Partially specified binding semantics may not work in all cases. Fundamentally, this restriction is due to the close coupling between texture samplers and texture coordinates in DirectX pixel shaders 1_X.

Binding Semantics for Uniform Data

If a binding semantic for a uniform parameter is not specified then the compiler will allocate one automatically. Scalar uniform parameters may be allocated to either the **`xyz`** or the **`w`** portion of a constant register depending on how they are used within the Cg program. When using the output of the compiler without the Cg runtime, you must set all values of a scalar uniform to the desired scalar value, not just the **`x`** component.

Table 34 summarizes the valid binding semantics for uniform parameters in the **`ps_1_X`** profiles. :

Table 34 DirectX Pixel Shader 1_X Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
`register(s0)–register(s3)` **`TEXUNIT0–TEXTUNIT3`**	Texture unit ***`N`***, where ***`N`*** is in range [0...3]. May be used only with uniform inputs with **`sampler*`** types.
`register(c0)–register(c7)` **`C0–C7`**	Constant register [0...7]

Binding Semantics for Varying Input/Output Data

The varying input binding semantics in the **ps_1_X** profiles are the same as the varying output binding semantics of the **vs_1_1** profile.

Varying input binding semantics in the **ps_1_X** profiles consist of **COLOR0**, **COLOR1**, **TEXCOORD0**, **TEXCOORD1**, **TEXCOORD2** and **TEXCOORD3**. These map to output registers in DirectX vertex shaders.

Table 35 summarizes the valid binding semantics for varying input parameters in the **ps_1_X** profiles. .

Table 35 DirectX Pixel Shader 1_X Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0 **COL, COL0**	Input color value v0
COLOR1 **COL1**	Input coor value v1
TEXCOORD0—TEXCOORD3 **TEX0—TEX3**	Input texture coordinates t0–t3

Additionally, the **ps_1_X** profiles allow **POSITION**, **FOG**, **PSIZE**, **TEXCOORD4**, **TEXCOORD5**, **TEXCOORD6**, and **TEXCOORD7** to be specified on varying inputs, provided these inputs are not referenced. This allows Cg programs to have the same structure specify the varying output of a **vs_1_1** profile program and the varying input of a **ps_1_X** profile program.

Table 36 summarizes the valid binding semantics for varying output parameters in the **ps_1_X** profile.

Table 36 DirectX Pixel Shader 1_X Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0 **COL, COL0**	Output color (float4)
DEPTH **DEPR**	Output depth (float)

The output depth value is special in that it may only be assigned a value in the **ps_1_3** profile, and must be of the form

```
...
float4 t = <texture addressing operation>;
float z = dot(texCoord<n>, t.xyz);
float w = dot(texCoord<n+1>, t.xyz);
depth = z / w;
...
```

Auxiliary Texture Functions

Because the capabilities of the texture addressing instructions are limited in DirectX pixel shader 1_X, a set of auxiliary functions are provided in these profiles that express the functionality of the more complex texture addressing instructions. These functions are merely provided as a convenience for writing **ps_1_X** Cg programs. The same result can be achieved by writing the expanded form of each function directly. Using the expanded form has the additional advantage of being supported on other profiles.

Table 37 summarizes these functions.

Table 37 DirectX Pixel Shader 1_X Auxiliary Texture Functions

Texture Function **Description**
offsettex2D(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m) **offsettexRECT(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m)**
Performs the following: **float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;** **return tex2D/RECT(tex, newst);** where **st** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, and **m** is the 2-D bump environment mapping matrix. This function can be used to generate the **texbem** instruction in all **ps_1_X** profiles.

Table 37 DirectX Pixel Shader 1_X Auxiliary Texture Functions

Texture Function **Description**
offsettex2DScaleBias(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m, uniform float scale, uniform float bias) **offsettexRECTScaleBias(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m, uniform float scale, uniform float bias)** Performs the following **float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;** **float4 result = tex2D/RECT(tex, newst);** **return result * saturate(prevlookup.z * scale + bias);** where **st** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **m** is the 2-D bump environment mapping matrix, **scale** is the 2-D bump environment mapping scale factor, and **bias** is the 2-D bump environment mapping offset. This function can be used to generate the **texbeml** instruction in all **ps_1_X** profiles.
tex1D_dp3(sampler1D tex, float3 str, float4 prevlookup) Performs the following **return tex1D(tex, dot(str, prevlookup.xyz));** where **str** are texture coordinates associated with sampler **tex**, and **prevlookup** is the result of a previous texture operation. This function can be used to generate the **texdp3tex** instruction in the **ps_1_2** and **ps_1_3** profiles.

Table 37 DirectX Pixel Shader 1_X Auxiliary Texture Functions

Texture Function Description

`tex2D_dp3x2(uniform sampler2D tex, float3 str, float4 intermediate_coord, float4 prevlookup)`
`texRECT_dp3x2(uniform samplerRECT tex, float3 str, float4 intermediate_coord, float4 prevlookup)`
Performs the following
`float2 newst = float2(dot(intermediate_coord.xyz, prevlookup.xyz), dot(str, prevlookup.xyz));`
`return tex2D/RECT(tex, newst);`
where
str are texture coordinates associated with sampler **tex**,
prevlookup is the result of a previous texture operation, and
intermediate_coord are texture coordinates associated with the previous texture unit.
This function can be used to generate the **texm3x2pad/texm3x2tex** instruction combination in all **ps_1_X** profiles.

`tex3D_dp3x3(sampler3D tex, float3 str, float4 intermediate_coord1, float4 intermediate_coord2, float4 prevlookup)`
`texCUBE_dp3x3(samplerCUBE tex, float3 str, float4 intermediate_coord1, float4 intermediate_coord2, float4 prevlookup)`
Performs the following
`float3 newst = float3(dot(intermediate_coord1.xyz, prevlookup.xyz), dot(intermediate_coord2.xyz, prevlookup.xyz), dot(str, prevlookup.xyz));`
`return tex3D/CUBE(tex, newst);`
where
str are texture coordinates associated with sampler **tex**,
prevlookup is the result of a previous texture operation,
intermediate_coord1 are texture coordinates associated with the '**n-2**' texture unit, and
intermediate_coord2 are texture coordinates associated with the '**n-1**' texture unit.
This function can be used to generate the **texm3x3pad/texm3x3pad/ texm3x3tex** instruction combination in all **ps_1_X** profiles.

Table 37 DirectX Pixel Shader 1_X Auxiliary Texture Functions

Texture Function Description
texCUBE_reflect_dp3x3(uniform samplerCUBE tex, float4 strq, float4 intermediate_coord1, float4 intermediate_coord2, float4 prevlookup)
Performs the following **float3 E = float3(intermediate_coord2.w, intermediate_coord1.w, strq.w);** **float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz), dot(intermediate_coord2.xyz, prevlookup.xyz), dot(strq.xyz, prevlookup.xyz));** **return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);** where **strq** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **intermediate_coord1** are texture coordinates associated with the **n-2** texture unit, and **intermediate_coord2** are texture coordinates associated with the **n-1** texture unit. This function can be used to generate the **texm3x3pad/texm3x3pad/ texm3x3vspec** instruction combination in all **ps_1_X** profiles.

Table 37 DirectX Pixel Shader 1_X Auxiliary Texture Functions

Texture Function **Description**
texCUBE_reflect_eye_dp3x3(uniform samplerCUBE tex, float3 str, float4 intermediate_coord1, float4 intermediate_coord2, float4 prevlookup, uniform float3 eye) Performs the following **float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz), dot(intermediate_coord2.xyz, prevlookup.xyz), dot(coords.xyz, prevlookup.xyz));** **return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);** where **strq** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **intermediate_coord1** are texture coordinates associated with the **n-2** texture unit, **intermediate_coord2** are texture coordinates associated with the **n-1** texture unit, and **eye** is the eye-ray vector. This function can be used to generate the **texm3x3pad/texm3x3pad/ texm3x3spec** instruction combination in all **ps_1_X** profiles.
tex_dp3x2_depth(float3 str, float4 intermediate_coord, float4 prevlookup) Performs the following **float z = dot(intermediate_coord.xyz, prevlookup.xyz);** **float w = dot(str, prevlookup.xyz);** **return z / w;** where **str** are texture coordinates associated with the ***n***th texture unit, **intermediate_coord**' are texture coordinates associated with the **n-1** texture unit, and **prevlookup** is the result of a previous texture operation. This function can be used in conjunction with the **DEPTH** varying out semantic to generate the **texm3x2pad**/**texm3x2depth** instruction combination in **ps_1_3**.

Examples

The following examples illustrate how a developer can use Cg to achieve DirectX pixel shader 1_X functionality.

Example 1:

```
struct VertexOut {
    float4 color     : COLOR0;
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
};

float4 main(VertexOut IN,
            uniform sampler2D diffuseMap,
            uniform sampler2D normalMap) : COLOR
{
    float4 diffuseTexColor = tex2D(diffuseMap,
IN.texCoord0.xy);
    float4 normal = 2 * (tex2D(normalMap, IN.texCoord1.xy) -
0.5);
    float3 light_vector = 2 * (IN.color.rgb - 0.5);
    float4 dot_result = saturate(dot(light_vector,
normal.xyz).xxxx);
    return dot_result * diffuseTexColor;
}
```

Example 2:

```
struct VertexOut {
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
    float4 texCoord2 : TEXCOORD2;
    float4 texCoord3 : TEXCOORD3;
};

float4 main(VertexOut IN,
            uniform sampler2D normalMap,
            uniform sampler2D intensityMap,
            uniform sampler2D colorMap) : COLOR
{
    float4 normal = 2 * (tex2D(normalMap, IN.texCoord0.xy) -
0.5);
    float2 intensCoord = float2(dot(IN.texCoord1.xyz,
normal.xyz),
                                dot(IN.texCoord2.xyz,
normal.xyz));
```

```
    float4 intensity = tex2D(intensityMap, intensCoord);
    float4 color = tex2D(colorMap, IN.texCoord3.xy);
    return color * intensity;
}
```

OpenGL NV_vertex_program 1.0 Profile (**vp20**)

The **vp20** Vertex Program profile is used to compile Cg source code to vertex programs for use by the **NV_vertex_program** OpenGL extension[7].

- **Profile Name:** **vp20**
- **How To Invoke:** Use the compiler option **-profile vp20**

This section describes the capabilities and restrictions of Cg when using the **vp20** profile.

Overview

The **vp20** profile limits Cg to match the capabilities of the **NV_vertex_program** extension. **NV_vertex_program** has the same capabilities as DirectX 8 vertex shaders, so the limitations that this profile places on the Cg source code written by the programmer is the same as the DirectX VS 1.1 shader profile[8].

Aside from the syntax of the compiler output, the only difference between the **vp20** Vertex Shader profile and the DirectX VS 1.1 profile is that the **vp20** profile supports two more outputs. The vertex-to-fragment connector can have the additional fields **BFC0** (for back-facing primary color) and **BFC1** (for back-facing secondary color).

Position Invariance

- The **vp20** profile supports position invariance, as described in the core language specification.
- The modelview-projection matrix must be specified using a binding semantic of **_GL_MVP**.

7. To understand the capabilities of NV_vertex_program and the code produced by the compiler using the vp20 profile, see the GL_NV_vertex_program extension documentation.
8. See "DirectX Vertex Shader 1.1 Profile (vs_1_1)" on page 222 for a full explanation of the data types, statements, and operators supported by this profile.

Data Types

This profile implements data types as follows:

- **`float`** data types are implemented as IEEE 32-bit single precision.
- **`half`** and **`double`** data types are implemented as **`float`**.
- **`int`** data type is supported using floating point operations, which add extra instructions for proper truncation for divides, modulos, and casts from floating point types.
- **`fixed`** or **`sampler*`** data types are not supported, but the profile does provide the minimal "partial support" that is required for these data types by the core language specification—that is, it is legal to declare variables using these types, as long as no operations are performed on the variables.

Bindings

Binding Semantics for Uniform Data

Table 38 summarizes the valid binding semantics for uniform parameters in the **`vp20`** profile.

Table 38 **`vp20`** Uniform Input Binding Semantics

Binding Semantics Name	Corresponding Data
`register(c0)-register(c95)` `C0-C95`	Constant register [0..95]. The aliases c0–c95 (lowercase) are also accepted. If used with a variable that requires more than one constant register (for example, a matrix), the semantic specifies the first register that is used.

Binding Semantics for Varying Input/Output Data

Table 39 summarizes the valid binding semantics for varying input parameters in the **vp20** profile.

One can also use **TANGENT** and **BINORMAL** instead of **TEXCOORD6** and **TEXCOORD7**. A second set of binding semantics, **ATTR0–ATTR15**, can also be used. The two sets act as aliases to each other.

Table 39 vp20 Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION, ATTR0	Input Vertex, Generic Attribute 0
BLENDWEIGHT, ATTR1	Input vertex weight, Generic Attribute 1
NORMAL, ATTR2	Input normal, Generic Attribute 2
COLOR0, DIFFUSE, ATTR3	Input primary color, Generic Attribute 3
COLOR1, SPECULAR, ATTR4	Input secondary color, Generic Attribute 4
TESSFACTOR, FOGCOORD, ATTR5	Input fog coordinate, Generic Attribute 5
PSIZE, ATTR6	Input point size, Generic Attribute 6
BLENDINDICES, ATTR7	Generic Attribute 7
TEXCOORD0-TEXCOORD7, ATTR8–ATTR15	Input texture coordinates (texcoord0-texcoord7), Generic Attributes 8-15
TANGENT, ATTR14	Generic Attribute 14
BINORMAL, ATTR15	Generic Attribute 15

Table 40 summarizes the valid binding semantics for varying output parameters in the **vp20** profile.

These binding semantics map to **NV_vertex_program** output registers. The two sets act as aliases to each other.

Table 40 vp20 Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
POSITION, HPOS	Output position
PSIZE, PSIZ	Output point size
FOG, FOGC	Output fog coordinate

Table 40 **vp20** Varying Output Binding Semantics (continued)

Binding Semantics Name	Corresponding Data
COLOR0, COL0	Output primary color
COLOR1, COL1	Output secondary color
BCOL0	Output backface primary color
BCOL1	Output backface secondary color
TEXCOORD0-TEXCOORD3, TEX0-TEX3	Output texture coordinates

The profile also allows **WPOS** to be present as binding semantics on a member of a structure of a varying output data structure, provided the member with this binding semantics is not referenced. This allows Cg programs to have the same structure specify the varying output of a **vp20** profile program and the varying input of an **fp30** profile program.

OpenGL NV_texture_shader and NV_register_combiners Profile (`fp20`)

The OpenGL NV_texture_shader and NV_register_combiners profile is used to compile Cg source code to the nvparse text format for the NV_texture_shader and NV_register_combiners family of OpenGL extensions[9].

- **Profile Name**: `fp20`
- **How To Invoke**: Use the compiler option `-profile fp20`

This document describes the capabilities and restrictions of Cg when using the `fp20` profile.

Overview

Operations in the `fp20` profile can be categorized as texture shader operations and arithmetic operations. Texture shader operations are operations which generate texture shader instructions, arithmetic operations are operations which generate register combiners instructions.

The underlying instruction set and machine architecture limit programmability in this profile compared to what is allowed by Cg constructs. Thus, this profile places additional restrictions on what can and cannot be done in a Cg program.

Restrictions

A Cg program in one of these profiles is limited to generating a maximum of four texture shader instructions and eight register combiner instructions. Since these numbers are quite small, users need to be very aware of this limitation while writing Cg code for these profiles.

The `fp20` profile also restricts when a texture shader operation or arithmetic operation can occur in the program. A texture shader operation may not have any dependency on the output of an arithmetic operation unless

- the arithmetic operation is a valid input modifier for the texture shader operation
- the arithmetic operation is part of a complex texture shader operation (which are summarized in the section "Auxiliary Texture Functions" on page 251)

9. For more details about the underlying instruction sets, their capabilities, and their limitations, please refer to the NV_texture_shader and NV_register_combiners extensions in the OpenGL Extensions documentation.

Modifiers

There are certain simple arithmetic operations that can be applied to inputs of texture shader operations and to inputs and outputs of arithmetic operations without generating a register combiner instruction. These operations are referred to as input modifiers and output modifiers.

Instead of generating a register combiners instruction, the arithmetic operation modifies the assembly instruction or source registers to which it is applied. For example, the following Cg expression

```
z = (x - 0.5 + y) / 2
```

could generate the following register combiner instruction (assuming **x** is in **tex0**, **y** is in **tex1**, and **z** is in **col0**)

```
rgb
 {
     discard = half_bias(tex0.rgb);
     discard = tex1.rgb;
     col0 = sum();
     scale_by_one_half();
 }
 alpha
 {
     discard = half_bias(tex0.a);
     discard = tex1.a;
     col0 = sum();
     scale_by_one_half();
 }
```

Table 41 summarizes how different NV_texture_shader and NV_register_combiners instruction set modifiers are expressed in Cg programs. For more details on the context in which each modifier is allowed and ways in which modifiers may be combined refer to the NV_texture_shader and NV_register_combiners documentation.

Table 41 NV_texture_shader and NV_register_combiners Instruction Set Modifiers

Instruction/Register Modifier	Cg Expression
scale_by_two()	`2*x`
scale_by_four()	`4*x`
scale_by_one_half()	`x/2`
bias_by_negative_one_half()	`x-0.5`

Table 41 NV_texture_shader and NV_register_combiners Instruction Set Modifiers (continued)

Instruction/Register Modifier	Cg Expression
bias_by_negative_one_half_scale_by_two()	`2*(x-0.5)`
unsigned(**reg**)	`saturate(x)` `(i.e. min(x, max(x, 1), 0))`
unsigned_invert(**reg**)	`1-saturate(x)`
half_bias(**reg**)	`x-0.5`
-**reg**	`-x`
expand(**reg**)	`2*(x-0.5)`

Language Constructs and Support

Data Types

In the **fp20** profile, operations occur on signed clamped floating-point values in the range -1 to 1. These profiles allow all data types to be used, but all operations are carried out in the above range. Refer to the NV_texture_shader and NV_register_combiners documentation for more details.

Statements and Operators

The **fp20** profile supports all of the Cg language constructs, with the following exceptions:

- Arbitrary swizzles are not supported (though arbitrary write masks are). Only the following swizzles are allowed
  .x/.r .y/.g .z/.b .w/.a
  .xy/.rg .xyz/.rgb .xyzw/.rgba
  .xxx/.rrr .yyy/.ggg .zzz/.bbb .www/.aaa
  .xxxx/.rrrr .yyyy/.gggg .zzzz/.bbbb .wwww/.aaaa
- Matrix swizzles are not supported.
- Boolean operators other than **<**, **<=**, > and **>=** are not supported. Furthermore, **<**, **<=**, **>** and **>=** are only supported as the condition in the **?:** operator.
- Bitwise integer operators are not supported.
- **/** is not supported unless the divisor is a non-zero constant or it is used to compute the depth output.

- `%` is not supported.
- Ternary `?:` is supported if the boolean test expression is a compile-time boolean constant, a uniform scalar boolean or a scalar comparison to a constant value in the range [-0.5, 1.0] (for example, `a > 0.5 ? b : c`).
- `do`, `for`, and `while` loops are supported only when they can be completely unrolled.
- arrays, vectors, and matrices may be indexed only by compile-time constant values or index variables in loops that can be completely unrolled.
- The `discard` statement is not supported. The similar but less general `clip()` function is supported.
- The use of an "allocation-rule-identifier" for input and output `structs` is optional.

Standard Library functions

Because the `fp20` profile has limited capabilities, not all of the Cg standard library functions are supported.

Table 42 presents the Cg standard library functions that are supported by this profile. See the standard library documentation for descriptions of these functions.

Table 42 Supported Standard Library Functions

dot(floatN, floatN)
lerp(floatN, floatN, floatN)
lerp(floatN, floatN, float)
tex1D(sampler1D, float)
tex1D(sampler1D, float2)
tex1Dproj(sampler1D, float2)
tex1Dproj(sampler1D, float3)
tex2D(sampler2D, float2)
tex2D(sampler2D, float3)
tex2Dproj(sampler2D, float3)
tex2Dproj(sampler2D, float4)
texRECT(samplerRECT, float2)

Table 42 Supported Standard Library Functions (continued)

texRECT(samplerRECT, float3)
texRECTproj(samplerRECT, float3)
texRECTproj(samplerRECT, float4)
tex3D(sampler3D, float3)
tex3Dproj(sampler3D, float4)
texCUBE(samplerCUBE, float3)
texCUBEproj(samplerCUBE, float4)

Note: The non-projective texture lookup functions are actually done as projective lookups on the underlying hardware. Because of this, the **`w`** component of the texture coordinates passed to these functions from the application or vertex program must contain the value 1.

Texture coordinate parameters for projective texture lookup functions must have swizzles that match the swizzle done by the generated texture shader instruction. While this may seem burdensome, it is intended to allow **`fp20`** profile programs to behave correctly under other pixel shader profiles.

Table 43 lists the swizzles required on the texture coordinate parameter to the projective texture lookup functions.

Table 43 Required Projective Texture Lookup Swizzles

Texture Lookup Function	Texture Coordinate Swizzle
tex1Dproj	.xw/.ra
tex2Dproj	.xyw/.rga
texRECTproj	.xyw/.rga
tex3Dproj	.xyzw/.rgba
texCUBEproj	.xyz/.rgba

Bindings

Manual Assignment of Bindings

The Cg compiler can determine bindings between texture units and uniform sampler parameters/texture coordinate inputs automatically. This automatic assignment is based on the context in which uniform sampler parameters and texture coordinate inputs are used together.

To specify bindings between texture units and uniform parameters/texture coordinates to match their application, all sampler uniform parameters and texture coordinate inputs that are used in the program must have matching binding semantics—for example, **`TEXUNIT<n>`** may only be used with **`TEXCOORD<n>`**. Partially specified binding semantics may not work in all cases. Fundamentally, this restriction is due to the close coupling between texture samplers and texture coordinates in the NV_texture_shader extension.

Binding Semantics for Uniform Data

If a binding semantic for a uniform parameter is not specified, then the compiler will allocate one automatically. Scalar uniform parameters may be allocated to either the **`xyz`** or the **`w`** portion of a constant register depending on how they are used within the Cg program. When using the output of the compiler without the Cg runtime, you must set all values of a scalar uniform to the desired scalar value, not just the **`x`** component.

Table 44 summarizes the valid binding semantics for uniform parameters in the **`fp20`** profile:

Table 44 **`fp20`** Uniform Binding Semantics

Binding Semantics Name	Corresponding Data
`register(s0)`—**`register(s3)`** **`TEXUNIT0`**—**`TEXTUNIT3`**	Texture unit ***`N`***, where ***`N`*** is in range [0...3]. May be used only with uniform inputs with **`sampler*`** types.

The **`ps_1_X`** profiles allow the programmer to decide which constant register a uniform variable will reside in by specifying the **`C<n>`**/**`register(c<n>)`** binding semantic. This is not allowed in the **`fp20`** profile since the **`NV_register_combiners`** extension does not have a single bank of constant registers. While the **`NV_register_combiners`** extension does describe constant registers, these constant registers are per-combiner stage and specifying bindings to them in the program would overly constrain the compiler.

Binding Semantics for Varying Input/Output Data

The varying input binding semantics in the **fp20** profile are the same as the varying output binding semantics of the **vp20** profile.

Varying input binding semantics in the **fp20** profile consist of **COLOR0**, **COLOR1**, **TEXCOORD0**, **TEXCOORD1**, **TEXCOORD2** and **TEXCOORD3**. These map to output registers in vertex shaders.

Table 45 summarizes the valid binding semantics for varying input parameters in the **fp20** profile.

Table 45 fp20 Varying Input Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0 COL, COL0	Input color value v0
COLOR1 COL1	Input coor value v1
TEXCOORD0—TEXCOORD3 TEX0—TEX3	Input texture coordinates t0–t3
FOGP FOG	Input fog color and factor

Additionally, the **fp20** profile allows **POSITION**, **PSIZE**, **TEXCOORD4**, **TEXCOORD5**, **TEXCOORD6**, and **TEXCOORD7** to be specified on varying inputs, provided these inputs are not referenced. This allows Cg programs to have the same structure specify the varying output of a **vp20** profile program and the varying input of a **fp20** profile program.

Table 46 summarizes the valid binding semantics for varying output parameters in the **fp20** profile.

Table 46 fp20 Varying Output Binding Semantics

Binding Semantics Name	Corresponding Data
COLOR, COLOR0 COL, COL0	Output color (float4)
DEPR DEPTH	Output depth (float)

The output depth value is special in that it may only be assigned a value of the form

```
...
float4 t = <texture shader operation>;
float z = dot(texCoord<n>, t.xyz);
float w = dot(texCoord<n+1>, t.xyz);
depth = z / w;
...
```

Auxiliary Texture Functions

Because the capabilities of the texture shader instructions are limited in NV_texture_shader, a set of auxiliary functions are provided in these profiles that express the functionality of the more complex texture shader instructions. These functions are merely provided as a convenience for writing **fp20** Cg programs. The same result can be achieved by writing the expanded form of each function directly. Using the expanded form has the additional advantage of being supported on other profiles.

Table 47 summarizes these functions.

Table 47 **fp20** Auxiliary Texture Functions

Texture Function **Description**
offsettex2D(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m) **offsettexRECT(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m)**
Performs the following: **float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;** **return tex2D/RECT(tex, newst);** where **st** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, and **m** is the offset texture matrix. This function can be used to generate the **offset_2d** or **offset_rectangle** NV_texture_shader instructions.

Table 47 **fp20** Auxiliary Texture Functions (continued)

Texture Function **Description**
offsettex2DScaleBias(uniform sampler2D tex, float2 st, float4 prevlookup, uniform float4 m, uniform float scale, uniform float bias) **offsettexRECTScaleBias(uniform samplerRECT tex, float2 st, float4 prevlookup, uniform float4 m, uniform float scale, uniform float bias)** Performs the following **float2 newst = st + m.xy * prevlookup.xx + m.zw * prevlookup.yy;** **float4 result = tex2D/RECT(tex, newst);** **return result * saturate(prevlookup.z * scale + bias);** where **st** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **m** is the offset texture matrix, **scale** is the offset texture scale, and **bias** is the offset texture bias. This function can be used to generate the **offset_2d_scale** or **offset_rectangle_scale** NV_texture_shader instructions.
tex1D_dp3(sampler1D tex, float3 str, float4 prevlookup) Performs the following **return tex1D(tex, dot(str, prevlookup.xyz));** where **str** are texture coordinates associated with sampler **tex**, and **prevlookup** is the result of a previous texture operation. This function can be used to generate the **dot_product_1d** NV_texture_shader instruction.

Table 47 **fp20** Auxiliary Texture Functions (continued)

Texture Function **Description**
tex2D_dp3x2(uniform sampler2D tex, float3 str, **float4 intermediate_coord, float4 prevlookup)** **texRECT_dp3x2(uniform samplerRECT tex, float3 str,** **float4 intermediate_coord, float4 prevlookup)** Performs the following **float2 newst = float2(dot(intermediate_coord.xyz, prevlookup.xyz),** **dot(str, prevlookup.xyz));** **return tex2D/RECT(tex, newst);** where **str** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, and **intermediate_coord** are texture coordinates associated with the previous texture unit. This function can be used to generate the **dot_product_2d** or **dot_product_rectangle** NV_texture_shader instruction combinations.
tex3D_dp3x3(sampler3D tex, float3 str, **float4 intermediate_coord1,** **float4 intermediate_coord2, float4 prevlookup)** **texCUBE_dp3x3(samplerCUBE tex, float3 str,** **float4 intermediate_coord1,** **float4 intermediate_coord2, float4 prevlookup)** Performs the following **float3 newst = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),** **dot(intermediate_coord2.xyz, prevlookup.xyz),** **dot(str, prevlookup.xyz));** **return tex3D/CUBE(tex, newst);** where **str** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **intermediate_coord1** are texture coordinates associated with the '**n-2**' texture unit, and **intermediate_coord2** are texture coordinates associated with the '**n-1**' texture unit. This function can be used to generate the **dot_product_3d** or **dot_product_cube_map** NV_texture_shader instruction combinations.

Table 47 **fp20** Auxiliary Texture Functions (continued)

Texture Function **Description**
texCUBE_reflect_dp3x3(uniform samplerCUBE tex, float4 strq, **float4 intermediate_coord1,** **float4 intermediate_coord2,** **float4 prevlookup)**
Performs the following **float3 E = float3(intermediate_coord2.w, intermediate_coord1.w,** **strq.w);** **float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),** **dot(intermediate_coord2.xyz, prevlookup.xyz),** **dot(strq.xyz, prevlookup.xyz));** **return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);** where **strq** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **intermediate_coord1** are texture coordinates associated with the **n-2** texture unit, and **intermediate_coord2** are texture coordinates associated with the **n-1** texture unit. This function can be used to generate the **dot_product_reflect_cube_map_eye_from_qs** NV_texture_shader instruction combination.

Table 47 **fp20** Auxiliary Texture Functions (continued)

Texture Function **Description**
texCUBE_reflect_eye_dp3x3(uniform samplerCUBE tex, **float3 str,** **float4 intermediate_coord1,** **float4 intermediate_coord2,** **float4 prevlookup,** **uniform float3 eye)**
Performs the following **float3 N = float3(dot(intermediate_coord1.xyz, prevlookup.xyz),** **dot(intermediate_coord2.xyz, prevlookup.xyz),** **dot(coords.xyz, prevlookup.xyz));** **return texCUBE(tex, 2 * dot(N, E) / dot(N, N) * N - E);** where **strq** are texture coordinates associated with sampler **tex**, **prevlookup** is the result of a previous texture operation, **intermediate_coord1** are texture coordinates associated with the **'n-2'** texture unit, **intermediate_coord2** are texture coordinates associated with the **n-1** texture unit, and **eye** is the eye-ray vector. This function can be used generate the **dot_product_reflect_cube_map_const_eye** NV_texture_shader instruction combination.
tex_dp3x2_depth(float3 str, float4 intermediate_coord, **float4 prevlookup)**
Performs the following **float z = dot(intermediate_coord.xyz, prevlookup.xyz);** **float w = dot(str, prevlookup.xyz);** **return z / w;** where **str** are texture coordinates associated with the ***n***th texture unit, **intermediate_coord**' are texture coordinates associated with the **n-1** texture unit, and **prevlookup** is the result of a previous texture operation. This function can be used in conjunction with the **DEPTH** varying out semantic to generate the **dot_product_depth_replace** NV_texture_shader instruction combination.

Examples

The following examples illustrate how a developer can use Cg to achieve NV_texture_shader and NV_register_combiners functionality.

Example 1:

```
struct VertexOut {
    float4 color     : COLOR0;
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
};

float4 main(VertexOut IN,
            uniform sampler2D diffuseMap,
            uniform sampler2D normalMap) : COLOR
{
    float4 diffuseTexColor = tex2D(diffuseMap,
IN.texCoord0.xy);
    float4 normal = 2 * (tex2D(normalMap, IN.texCoord1.xy) -
0.5);
    float3 light_vector = 2 * (IN.color.rgb - 0.5);
    float4 dot_result = saturate(dot(light_vector,
normal.xyz).xxxx);
    return dot_result * diffuseTexColor;
}
```

Example 2:

```
struct VertexOut {
    float4 texCoord0 : TEXCOORD0;
    float4 texCoord1 : TEXCOORD1;
    float4 texCoord2 : TEXCOORD2;
    float4 texCoord3 : TEXCOORD3;
};

float4 main(VertexOut IN,
            uniform sampler2D normalMap,
            uniform sampler2D intensityMap,
            uniform sampler2D colorMap) : COLOR
{
    float4 normal = 2 * (tex2D(normalMap, IN.texCoord0.xy) -
0.5);
    float2 intensCoord = float2(dot(IN.texCoord1.xyz,
normal.xyz),
                                dot(IN.texCoord2.xyz,
normal.xyz));
    float4 intensity = tex2D(intensityMap, intensCoord);
```

```
    float4 color = tex2D(colorMap, IN.texCoord3.xy);
    return color * intensity;
}
```