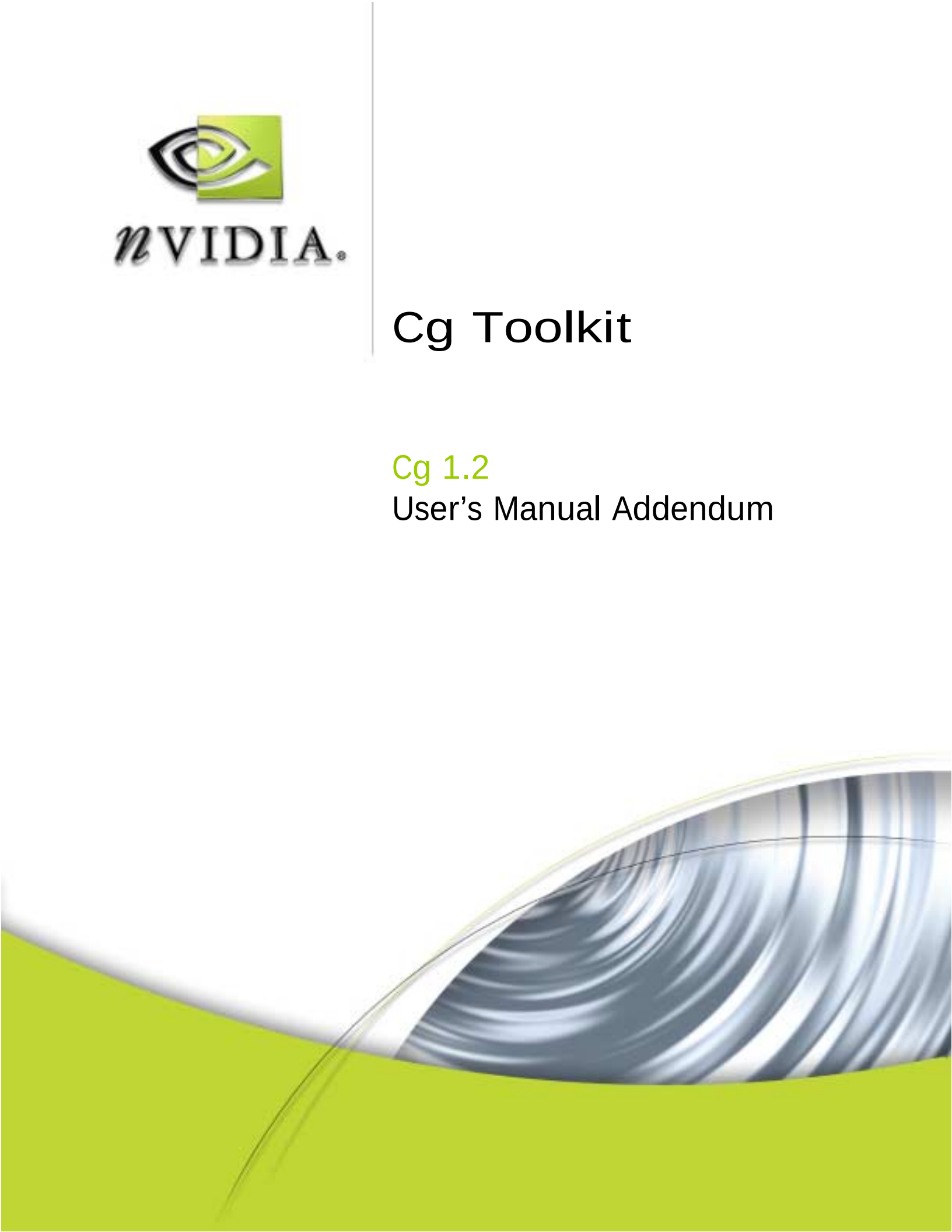

Cg Toolkit

Cg 1.2
User's Manual Addendum

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation.

Microsoft, Windows, the Windows logo, and DirectX are registered trademarks of Microsoft Corporation.

OpenGL is a trademark of SGI.

Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Cg 1.2
User's Manual Addendum

Introduction

Programs written using current GPU programming languages tend to be rather inflexible. For example, computing a pixel color might involve looping over some number of different material layers, or some number of different light sources. Current languages require that the exact number of materials or light sources be specified when the program is written, rather than dynamically when the program is being used by an application. Similarly, the exact implementation of each material layer or light source must be specified when the program is written – there is no convenient way to "plug in" a different kind of material or light source at application runtime.

The 1.2 release of Cg introduces a number of new language and runtime features that make it possible for developers to write modular Cg constructs that can be "wired together" at application runtime to construct novel Cg programs. For example, using Cg 1.2 it is possible to create families of modules that share a common interface, each member of which has a different implementation. This ability makes it easy for applications to construct Cg material trees on the fly, to change the number or type of texture maps applied to an object at application runtime, and so on.

In addition, Cg 1.2 supports the sharing of application-created parameter instances between multiple Cg programs. Parameter sharing makes it possible for an application to dynamically change the value of a single shared parameter – for example, the position or color of a light source -- and for that change automatically to propagate to all Cg programs that use the parameter.

Cg 1.2 also allows you to mark uniform program parameters as compile-time constants (literals) prior to program compilation. This feature can be used to "bake" parameter values into the compiled Cg program, which often produces much more efficient compiled code.

This addendum to the Cg User's Manual provides an overview of these and other new language and runtime features introduced in release 1.2 of the Cg compiler and runtime.

Language Additions

The 1.2 release of the Cg Toolkit introduces three new features to the Cg language: structure member functions, interfaces, and unsized arrays.

Structure Methods

Structures may define member functions in addition to member variables. Member functions provide a convenient way of encapsulating helper functions associated with the data in the structure, or as a means of describing the behavior of a data "object".

Structure member functions are declared and defined within the body of the structure definition:

```
struct Foo {
    float val;
    float helper(float x) {
        return val + x;
    }
};
```

Member functions may reference their arguments or the member variables of the structure in which they are defined. The result of referring to a variable outside the scope of the enclosing structure (e.g., global variables) is undefined; instead, passing such variables as arguments to member functions that need them is recommended.

Member functions are invoked using the usual "**.**" notation:

```
float4 main(uniform Foo myfoo, uniform float myval) : COLOR {
    return myfoo.helper(myval);
}
```

Note that in the current release, members variables must be declared before member functions that reference them. In addition, member functions may not be overloaded based on profile in this release.

Interfaces

Cg 1.2 supports interfaces, a language construct found in other languages, including Java and C# (or in C++ as pure virtual classes). Interfaces provide a means of abstractly describing the member functions a particular structure provides, without specifying how those functions are implemented. When used in conjunction with parameter instantiation, this abstraction makes it possible to "plug in" any structure that implements a given interface into a program – even if the structure was not known to the author of the original program.

An interface declaration describes a set of member functions that a structure must define in order to implement the named interface. Interfaces contain only function prototype definitions. They do not contain actual function implementations or data members. For example, the following example defines an interface named **`Light`** consisting of two method, **`illuminate`** and **`color`**:

```
interface Light {
    float3 illuminate(float3 P, out float3 L);
    float3 color(void);
};
```

A Cg structure may optionally implement an interface. This is signified by placing a "**`:`**" and the name of the interface after the name of the structure being defined.

The methods required by the interface must be defined within the body of the structure. For example:

```
struct SpotLight : Light {
    sampler2D shadow;
    samplerCUBE distribution;
    float3 Plight, Clight;
    float3 illuminate(float3 P, out float3 L) {
        L = normalize(Plight - P);
        return Clight * tex2D(shadow, P).xxx *
            texCUBE(distribution, L).xyz;
    }
    float3 color(void) {
        return Clight;
    }
};
```

Here, the **SpotLight** structure is defined, which implements the **Light** interface. Note that the **illuminate()** and **color()** methods are defined within the body of the structure, and that their implementations are able to reference data members of the **SpotLight** structure (e.g., **Plight**, **Clight**, **shadow**, and **distribution**).

Function parameters, local variables, and global variables all may have interface types. Interface parameters to top-level functions (e.g., main) must be declared as **uniform**.

A structure that implements a particular interface may be used wherever that its interface type is expected. For example:

```
float3 myfunc(Light light) {
    float3 result = light.illuminate(...);
    ...
}

float4 main(uniform SpotLight spot) {
    float3 color = myfunc(spot);
    ...
 }
```

Here, the **SpotLight** variable spot may be used as a generic **Light** in the call to **myfunc**, because **SpotLight** implements the **Light** interface.

It is possible to declare a local variable of an interface type. However, a concrete structure must be assigned to that variable before any of the interface's methods may be called. For example:

```
  Light mylight;
  SpotLight spot;
  float3 color;
  ... /* initialize spot */ ...

  color = mylight.illuminate(...);
  mylight = spot;
  color = mylight.illuminate(...); // OK
```

Under all current profiles, the concrete implementation of all interface method calls must be resolvable at compile time. There is no dynamic 'run-time' determination of which implementation to call under any current profile.

See the "interfaces_ogl" example, included in the Cg 1.2 distribution, for an example of the use of interfaces.

Unsized Arrays

Cg 1.2 supports "unsized" arrays – arrays with one or more dimension having no specified length. This makes it possible to write Cg functions that operate on arrays of arbitrary size.

For example:

```
float myfunc(float vals[]) {
    ...
}
```

Here, **myfunc** is declared to be a function of a single parameter, **vals**, which is a one-dimensional array of floats. However, the length of the **vals** array is not specified.

The effect of this declaration is that any following call to **myfunc()** that passes a one-dimensional array of floats of any size will resolve to the declared function. For example:

```
float myfunc(float vals[]) {
    ...
}

float4 main(...) {
  float vals1[2];
  float vals2[76];
  ...
  float myval1 = myfunc(vals1);  // match
  float myval2 = myfunc(vals2);  // match
  ...
}
```

The actual length of an array parameter (sized or unsized) may be queried via the **.length** pseudo-member:

```
float myfunc(float vals[]) {
    float sum = 0;
    for (int i = 0; i < vals.length; i++) {
        sum += vals[i];
    }
    return sum;
}
```

The size of a particular dimension of a multidimensional array may be queried by dereferencing the appropriate number of dimensions of the array. For example, **vals2d[0].length** gives the length of the second dimension of the two-dimensional **vals2d** array:

```
float myfunc(float vals2d[][]) {
    float sum = 0;
    for (int i = 0; i < vals2d.length; i++) {
        for (int j = 0; i < vals2d[0].length; j++) {
            sum += vals[i][j];
        }
    }
    return sum;
}
```

If the length of any dimension of an array parameter is specified, that parameter will only match calls with variables whose corresponding dimension is of the specified length. For example:

```
float func(float vals[6][]) {
  ...
}

float4 main(...) {
    float v1[6][7];
    float v2[5][11];
    ...
    float myv1 = func(vals1);  // match: 6 == 6
    float myv2 = func(vals2);  // no match: 5 != 6
}
```

Note that unsized arrays may only be declared as function parameters -- they may not be declared as variables. Furthermore, in all current profiles, the actual array length and address calculations implied by array indexing must be known at compile time.

Unsized array parameters of top-level functions (e.g. main) may be connected to sized arrays that are created in the runtime, or their size may be set directly for convenience. See the **`cgSetArraySize`** manual in the Cg 1.2 core runtime documentation for details.

Notes & caveats

- A structure member variable must currently be declared before any method that uses that variable is declared.
- There is no inheritance per se in Cg: a structure may not inherit from another structure.
- Structures may only implement a single interface.
- Interfaces cannot be extended or combined.

These limitations may be addressed in future releases.

Note that although there is no structure inheritance, it is possible to define a 'default' implementation of a particular interface method. The default implementation can be defined as a global function, and structures that implement that interface may then call this default method via a wrapper.

Note also that interface and structure parameters of top-level functions (e.g. main) may be connected to structures that are created in the runtime. See **Cg 1.2 Runtime Additions**, below, for more details.

Cg 1.2 Runtime Additions

Version 1.2 of the Cg runtime adds many new capabilities to the existing set of functionality. The most notable new capabilities are program recompilation, shared parameters, support for user-defined types, and unsized array support. Each of these new features is discussed in the remainder of this document. For details on a specific entry point, please see the online documentation.

Runtime Program Recompilation

Because current hardware profiles do not efficiently support branching, much of the new Cg 1.2 functionality requires that the application perform certain kinds of operations on a Cg program before the program is compiled. For example, the application often must specify sizes for unsized array parameters, or connect parameter instances to interface program parameters.

In prior releases of Cg, a Cg program could always be automatically compiled when first loaded by the Cg runtime. In Cg 1.2, however, it is possible – and often necessary -- for a program to exist in an "uncompiled" state. Once the actual size of all unsized array parameters is specified, all interface parameters are connected to structure instances, and any desired uniform parameters have been marked as literals, the program can then be compiled.

The Cg 1.2 runtime environment allows for either automatic or manual recompilation of programs. Recompilation of a program is necessary when the program is first created, or when it enters an uncompiled state. A program may enter an uncompiled state for a variety of reasons, including:

- Changing variability of parameters

  Parameters may be changed from uniform variability to literal variability (compile time constant). See the **`cgSetParameterVariability`** manual page for more information.

- Changing value of literal parameters

  Changing the value of a literal parameter will require recompilation since the value is used at compile time. See the **`cgSetParameter`** and **`cgSetMatrixParameter`** manual pages for more information.

- Resizing unsized arrays

  Changing the length of a parameter array may require recompilation depending on the capabilities of the program profile. See the **`cgSetArraySize`** and **`cgSetMultiDimArraySize`** manual pages for more information.

- Connecting structures to interface parameters

  Structure parameters can be connected to interface program parameters to control the behavior of the program. Changing these connections require recompilation on all current profiles. See the **`cgConnectParameter`** manual page and the Interfaces section of this document for more details.

When a program enters an uncompiled state, it is automatically unloaded and unbound. In order to be used again, the program must be recompiled (either automatically or manually – see below), and then reloaded and rebound.

Recompilation can be performed manually by the application via **`cgCompileProgram()`** or automatically by the runtime. Recompilation behavior is controlled via the entry point **`cgSetAutoCompile`**:

```
    void cgSetAutoCompile(CGcontext ctx, CGenum flag);
```

Here, **`flag`** may be one of the following enumerants:

`CG_COMPILE_MANUAL`

In this mode, the application is responsible for manually recompiling a program. The application may check to see if a program requires recompilation with the entry point **`cgIsProgramCompiled`**. The program may then be compiled via **`cgCompileProgram.`** This mode provides the application with the most control over how and when program recompilation occurs.

`CG_COMPILE_IMMEDIATE`

In this mode, the Cg runtime will force recompilation automatically and immediately when a program enters an uncompiled state.

`CG_COMPILE_LAZY`

This mode is similar to **`CG_COMPILE_IMMEDIATE`**, but will delay program recompilation until the program object code is needed. The advantage of this method is the reduction of extraneous recompilations. The disadvantage is that compile time errors will not be encountered when the program enters an uncompiled state, but will instead be encountered at some later time (most likely when the program is loaded or bound).

User-Defined Types

In the Cg language, named types may be defined in a program when declaring structure and interface types. For example, if the following Cg code is included in the source to a CGprogram created via **`cgCreateProgram*()`**, the types **`MyInterface`** and **`MyStruct`** will be added to the resulting **`CGprogram`**.

```
interface MyInterface {
    float SomeMethod(float x);
};

struct MyStruct : MyInterface {
    float Scale;

    SomeMethod(float x) {
        return(Scale * x);
    }
};
```

The **`CGtype`** associated with a user-defined named type can be retrieved using **`cgGetNamedUserType`**:

```
CGtype cgGetNamedUserType(CGprogram program, const char *name);
```

Once the **CGtype** has been retrieved, it may be used to create an instance of the structure using **cgCreateParameter**. The instance may then be connected to a program parameter of the parent type (in the above example this would be **MyInterface**) using **cgConnectParameter**.

In order to preserve backward compatibility for existing applications, calling **cgGetParameterType** on such a parameter will return the enumerant **CG_STRUCT**. In order to obtain the unique enumerant of the user-defined type, the following entry point should be used:

```
CGtype cgGetParameterNamedType(CGparameter param);
```

The parent types of a given named type may be obtained with the following entry points:

```
int cgGetNumParentTypes(CGtype type);
CGtype cgGetParentType(CGtype type, int index);
```

All of the user-defined types associated with a program may be obtained with the following entry points:

```
int cgGetNumUserTypes(CGprogram program);
CGtype cgGetUserType(CGprogram program, int index);
```

Note that the runtime considers interface program parameters as if they were structure parameters with no concrete data or function members.

Type Equivalency

If a program containing a user-defined type is created in a context that already contains another program that defines a user type with the same name, the two type definitions are compared. If both type definitions are found to be equivalent, the **CGtype** enumerant associated with the user type in the new program will be identical to that of the identical user type in the existing program. If the types are not equivalent, the new type will be assigned a unique **CGtype**. In this way, type equivalency of parameters shared between multiple programs can be assured simply by comparing **CGtype** enumerants.

In order for a two types to be considered equivalent, they must meet the following requirements:

- The type names must match

  Both types must have the exact same name.

- The parents types, if any, must match

  If the type is a structure, both must either not implement an interface, or both implement interfaces that are type-equivalent.

- The member variables and methods must match

  They must both have the exact same member variables and methods. The order and name of the variables must match exactly, and the order and name of the methods must match. The signature of the methods, including argument and return types, must be identical.

Type equivalency is useful when using shared parameters instances with multiple programs by connecting them with **cgConnectParameter()**. Please see the following section for more information.

Shared Parameter Instances

The 1.2 release of the Cg runtime supports the creation of instances of any type of concrete parameter (e.g. built-in types, user-defined structures) within a Cg context. A parameter instance may be connected to any number of compatible parameters, including the top-level parameters of any programs within the context.

Note that in this release, **connecting instances to program parameters is not supported when using the Cg D3D runtime libraries**. This functionality is currently only supported when using the Cg OpenGL runtime library.

When an instance is connected to another parameter, the second parameter will then inherit its value(s) from the instance. Furthermore, if the variability of the second parameter has not been explicitly set by a call to **`cgSetParameterVariability,`** or if it has been set to **`CG_DEFAULT,`** its variability will also be inherited from the instance.

This ability to create and easily manage context-global parameters provides a powerful means of creating parameter "trees", and of sharing data and user-defined objects between multiple Cg programs.

Creating Parameter Instances

Parameter instances are associated with a **`CGcontext`**, rather than a **`CGprogram`**. They may be created with the following entry points:

```
CGparameter cgCreateParameter(CGcontext ctx, CGtype type);
CGparameter cgCreateParameterArray(CGtype type, int length);
CGparameter cgCreateParameterMultiDimArray(CGtype type,
                                           int dim,
                                           const int *lengths);
```

Only parameters of concrete types may be created. In particular, parameters of interface (abstract) types may not be created. By default, a created parameter has uniform variability, and undefined values.

Deleting Parameter Instances

Parameter instances may be deleted using:

```
void cgDeleteParameter(CGparameter param);
```

When an instance is deleted, all parameters it was connected to are disconnected; see below for more details.

Connecting Parameters

Once created, a parameter instance may be connected to any number of program parameters, or other instance parameters, with **`cgConnectParameter()`**:

```
void cgConnectParameter(CGparameter from, CGparameter to);
```

Where **from** is the instance and **to** is the program parameter or other instance. Two parameters may be connected if they are of an identical type, or if **to** is an interface type that is implemented by **from**.

Once a parameter has been connected "to", its value should no longer be set directly. Instead, the value should be set indirectly by setting the value of the **from** parameter using **cgSetParameter** or **cgSetMatrixParameter**.

When setting the values of parameter instances, the entry points **cgSetParameter** and **cgSetMatrixParameter** must be used instead of the graphics-API-specific entry points (e.g,. **cgGLSetParameter3f()**). These Cg core runtime entry points may also optionally be used instead of the API-dependent runtime equivalents once a program is loaded or bound using the API-dependent runtime. There is no particular advantage to using the core runtime functions over the equivalent functions in the API-dependent runtimes when dealing with program parameters.

A shared parameter can be disconnected from a program parameter using:

```
void cgDisconnectParameter(CGparameter param);
```

Connecting Structures to Interface parameters

Parameters of a user-defined structure type may be connected to a particular interface parameter as long as the structure type implements that interface. **cgIsParentType,** coupled with **cgGetParameterNamedType,** may be used to determine parenthood at runtime.

When a structure parameter is connected to an interface parameter, copies of any child (i.e. member) variables under the source structure parameters are automatically created as children of the destination parameter by the runtime. Under most circumstances, these member variables copies can be ignored by the application, since their values and variability are automatically set by the Cg runtime. However, it may be useful to query such a copied parameter for its underlying resource, for example, in some situations.

An instance of a structure whose type is defined in one program may be connected to parameter(s) of other programs provided that all of the programs define the source structure type and destination interface type equivalently. See Type Equivalency, above, for more details. If they types are not equivalent, **cgConnectParameter** will generate a runtime error. The following example illustrates structure-to-interface connection by creating three programs, all of which define a type named **Foo,** with one program's definition differing from the other two.

```
interface Foo {
    float Val(float x);
};

struct Bar : MyInterface {
    float Scale;
    float Val(float x) { return(Scale * x);
};

float4 main(Foo foo) : COLOR {
   return(foo.Val(.2).xxxx);
}
```

Listing 1: Cg Program 1

```
interface Foo {
    float Val(float x);
};

struct Bar : MyInterface {
    float Scale;
    float Val(float x) { return(Scale * x);
};

float4 main(Foo foo) : COLOR {
   return(foo.Val(.3).xxxx);
}
```

Listing 2: Cg Program 2

```
interface Foo {
    half Val(half x);
};

struct Bar : MyInterface {
    float Scale;
    half Val(half x) { return(Scale * x);
};

float4 main(Foo foo) : COLOR {
    return(foo.Val(.5).xxxx);
}
```

Listing 3: Cg Program 3

Notice that both **Cg Program 1** and **Cg Program 2** define the **`Val()`** method of the **`Foo`** and **`Bar`** types using the **`float`** type, whereas **`Cg Program 3`** does so using the **`half`** type. As a result, the **`Foo`** and **`Bar`** types defined in **`Cg Program 3`** are not equivalent to types in the other two programs, even though the types have the same names.

The following C program creates all three of the above Cg programs and connects shared parameter instances to their input parameters:

```
static CGprogram CreateProgram(const char *program_str) {
  return cgCreateProgram(Context, CG_SOURCE,
                         program_str, CG_PROFILE_ARBFP1,
                         "main", NULL);
}

int main(int argc, char *argv[]) {
  CGContext Context;
  CGprogram Program1, Program2, Program3;
  CGparameter Foo1, Foo3, Scale;

  // Disable automatic compilation, since the
  // programs cannot be compiled until concrete structs
  // are connected to each program's interface parameters.
  Context = cgCreateContext();
  cgSetAutoCompile(Context, CG_COMPILE_MANUAL);

  // Create the programs
  Program1 = CreateProgram(Program1String);
  Program2 = CreateProgram(Program2String);
  Program3 = CreateProgram(Program3String);

  // Create two shared parameters,
  // one of the Foo type from Program1, and
  // one of the Foo type from Program3.
  Foo1 = cgCreateParameter(cgGetNamedUserType(Program1, "Foo"));
  Foo3 = cgCreateParameter(cgGetNamedUserType(Program3, "Foo"));

// Connect the same shared parameter to Program1 and Program2
  cgConnectParameter(Foo1, cgGetNamedParameter(Program1, "foo"));
  cgConnectParameter(Foo1, cgGetNamedParameter(Program2, "foo"));

  // The following would generate an error, because the type
  // of the Foo1 parameter is not equivalent to the type "Foo"
  // from Program3.
  // cgConnectParameter(Foo1,
  //                    cgGetNamedParameter(Program3, "foo"));

  cgConnectParameter(Foo3, cgGetNamedParameter(Program3, "foo"));

  // Now we can compile all three programs.
  cgCompileProgram(Program1);
  cgCompileProgram(Program2);
  cgCompileProgram(Program3);

// The Scale member of both Foo struct types have identical
// types (CG_FLOAT); we can create a shared parameter
// and connect it to the Scale members of the previously-
// created structs.  Setting its value affects all 3 programs.
  Scale = cgCreateParameter(Context, CG_FLOAT);
  cgConnectParameter(Scale, cgGetNamedStructParameter(Foo1,
"Scale"));
  cgConnectParameter(Scale, cgGetNamedStructParameter(Foo2,
"Scale"));
  cgSetParameter1f(Scale, 0.7f);
}
```

Cg OpenGL Runtime Changes

In prior release of Cg, applications using the CgGL runtime were responsible for enabling the texture parameters associated a program. In the 1.2 release of Cg, the CgGL runtime can automatically enable all texture parameters associated with a program when that program is bound via **`cgGLBindProgram().`**

This behavior may be controlled on a per-context basis using:

```
void cgGLSetManageTextureParameters(CGcontext ctx, CGbool flag);
```

Texture parameter management is disabled by default when a context is created.

Compiler Changes

A new profile option, **`MaxTexIndirections`**, is now supported under the arbfp1 profile. When the specified value is less than the maximum allowed number of texture instructions (specified using the **`NumTexInstructionSlots`** profile option), the compiler will attempt to minimize the number of texture indirections in the generated assembly program. If the program requires more than the indicated number of texture indirections, the compiler will issue an error.

As with other profile-specific options, calling **`cgGLSetOptimalOptions()`** before invoking **`cgCompileProgram*()`** will automatically set these profile options to values appropriate for the active GPU and driver.