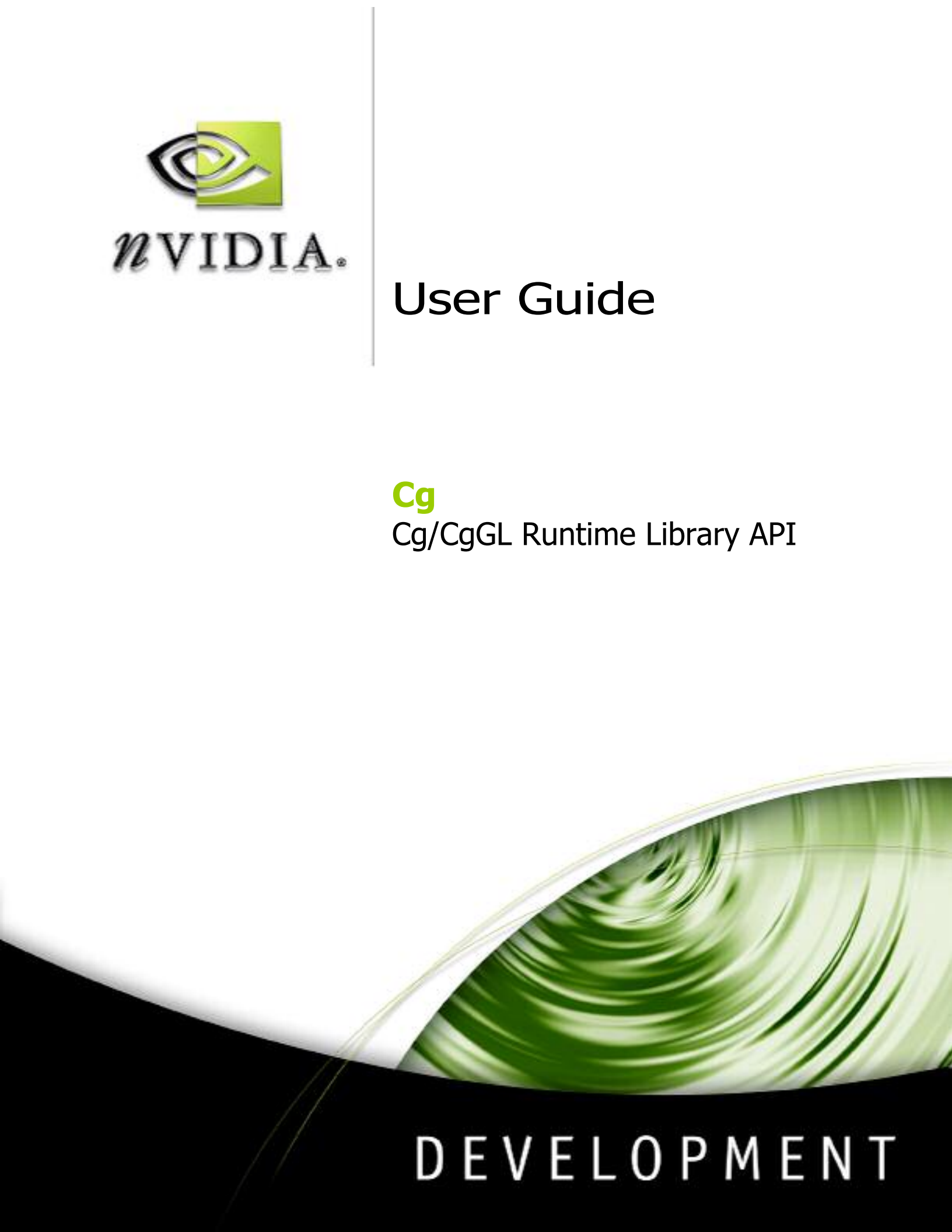
NVIDIA.
User Guide
Cg
Cg/CgGL Runtime Library API
DEVELOPMENT

Table of Contents

List of Tables

Note: Please send all feedback to cgsupport@nvidia.com.

Using the Old API

The original beta runtime API is now deprecated, but can still be used by defining **`CG_DEPRECATED_API`** before including **`cg.h`** or **`cgGL.h`**.

Types

All the Cg types have been renamed. Table 1 lists the old Cg types and their equivalents in the new runtime.

Table 1. Deprecated API vs New API

Deprecated API	New API
cgContext	CGcontext
cgProgramIter	CGprogram
cgBindIter	CGparameter
cgProfileType	CGprofile
cgError	CGerror

Also be aware that you no longer need to declare pointers to your variables as the types are already pointers. For example, if you had previously declared your Cg context like this:

```
cgContext *context;
```

it would now be declared like this:

```
CGcontext context;
```

Setting up an Error Callback

The new runtime now has the ability to set up an error callback function that gets executed whenever an error occurs within the runtime. This is useful for catching any errors that might occur without having to check the return value of every runtime function that you use.

Following is an example of error callback that displays the last error as well as the listing from the compiler, and then exits:

```
void cgErrorCallback(void)
{
    CGerror LastError = cgGetError();

    if(LastError)
    {
        printf("%s\n\n", cgGetErrorString(LastError));
        printf("%s\n", cgGetLastListing(context));
        printf("Cg error, exiting...\n");

        exit(0);
    }
}
```

To register this error callback we call the **cgSetErrorCallBack()** function and pass in our callback function:

```
cgSetErrorCallback(cgErrorCallback);
```

This should be done before calling any Cg runtime functions to ensure that all errors are caught.

Creating a Cg Context

This step hasn't changed; you just call **cgCreateContext()** and it returns a handle to your Cg context.

Example:

```
context = cgCreateContext();
```

If **cgCreateContext()** fails, it returns NULL. If you have registered an error callback, it will get executed.

The Cg context returned by **cgCreateContext()** can be destroyed using the **cgDestroyContext()** function.

Adding Programs to the Cg Context

The **cgAddProgram()** and **cgAddProgramFromFile()** functions have been renamed to **cgCreateProgram()** and **cgCreateProgramFromFile()**. Both functions return a program object, so the **cgProgramByName()** function has been removed.

Function Definitions

```
CGprogram cgCreateProgram(CGcontext ctx,
                          CGenum program_type,
                          const char *program,
                          CGprofile profile,
                          const char *entry,
                          const char **args);

CGprogram cgCreateProgramFromFile(CGcontext ctx,
                                  const char *program_file,
                                  CGprofile profile,
                                  const char *entry,
                                  const char **args);
```

Both functions can now add precompiled shaders by specifying **CG_OBJECT** for the **program_type** parameter. When passing in a Cg source program you would just specify **CG_SOURCE** for this parameter.

Example:

```
string filename = "simpleVP.cg";
program = cgCreateProgramFromFile(context, // Cg context
                                  CG_SOURCE,       // cg program
                                  filename.data(), // filename of the
                                                   // program
                                  CG_PROFILE_VP20, // NV_vertex_program
                                                   // profile
                                  NULL,            // use default "main"
                                                   // entry
                                  NULL);           // no arguments for
                                                   // the compiler
```

If the function fails with a **CG_COMPILE_ERROR**, you can call **cgGetLastListing()** to get the output from the compiler to see why it's failing.

The program object returned by **cgCreateProgram()** can be destroyed using the **cgDestroyProgram()** function.

The profile enumerants have all been changed to match the enumerant naming conventions of the new runtime, as well as the profiles that the compiler accepts. Table 2 lists the old Cg profile types and their equivalents in the new runtime.

Table 2. Profile Types

Deprecated API	New API
cgVertexProfile	CG_PROFILE_VP20
cgVertex30Profile	CG_PROFILE_VP30
cgARBVertexProfile	CG_PROFILE_ARBVP1
cgFragment20Profile	CG_PROFILE_FP20
cgFragmentProfile	CG_PROFILE_FP30
cgDX8Vertex	CG_PROFILE_DX8VS
cgDX8Pixel	CG_PROFILE_DX8PS

Retrieving Program Parameters

The **cgGetBindByName()** function has been renamed to **cgGetNamedParameter()**, and updated to use the new Cg types. This function will return NULL when a given parameter cannot be found.

Function Definition

```
CGparameter cgGetNamedParameter(CGprogram prog, const
char *name);
```

Example:

```
CGparameter constantsParam;
CGparameter modelViewProjParam;
constantsParam = cgGetNamedParameter(program, "Constants");
modelViewProjParam = cgGetNamedParameter(program,
                                         "ModelViewProj");

if (!constantsParam || !modelViewProjParam)
{
    printf("Unable to retrieve program parameters,
           exiting...\n");
    exit(0);
}
```

Iterating Through Program Parameters

The old API didn't have the concept of array or struct parameters. For example, imagine if we had the following structs:

```
struct Foo
{
  float A;
  float B;
};

struct Bar
{
  Foo C[2];
};
```

and a uniform program parameter:

```
uniform Bar MyBar[2];
```

Iterating with the old API would return the following sequence of parameters:

```
MyBar[0].C[0].A
MyBar[0].C[0].B
MyBar[0].C[1].A
MyBar[0].C[1].B
MyBar[1].C[0].A
MyBar[1].C[0].B
MyBar[1].C[1].A
MyBar[1].C[1].B
```

In the new API, the above parameters are all considered leaf parameters. The equivalent to **cgGetNextBind()** is the combination of **cgGetFirstLeafParameter()** and **cgGetNextLeafParameter()**. Previously, you could get the first parameter in a program by calling **cgGetNextBindIter()** with the second parameter passed in as NULL. Now we have **cgGetFirstLeafParameter()** which explicitly returns this. We have also added the **name_space** parameter to allow for iteration through different namespaces in the future. Currently, the only supported namespace is **CG_PROGRAM**.

The new API now allows you to retrieve parameters for arrays and structs as well as the individual items within an array or struct. For example the new API would return something like this for the above struct and array definitions:

```
MyBar
MyBar[1]
MyBar[1].C
```

Using the **cgGetFirstParameter()** and **cgGetNextParameter()** function calls. These functions differ from **cgGetFirstLeafParameter()** and **cgGetNextLeafParameter()** in that they iterate through all parameters including structs and arrays.

Parameter Recursion Functions

```
CGparameter cgGetFirstParameter(CGprogram prog,
                                CGenum name_space);
CGparameter cgGetNextParameter(CGparameter current);
CGparameter cgGetFirstStructParameter(CGparameter param);
CGparameter cgGetArrayParameter(CGparameter aparam,
                                int index);
int cgGetArraySize(CGparameter param, int dimension);
```

Example code that recurses through all program parameters:

```
void RecurseParamsInProgram(CGprogram prog)
{
  RecurseParams(cgGetFirstParameter(prog, CG_PROGRAM));
}

void RecurseParams(CGparameter param)
{
  if(!param) return;

  do
  {
    switch(cgGetParameterType(param))
    {
      case CG_STRUCT :
        RecurseParams(cgGetFirstStructParameter(param));
        break;

      case CG_ARRAY :
      {
        int ArraySize = cgGetArraySize(param, 0);
        int i;
```

```
        for(i=0; i < ArraySize; ++i)
          RecurseParams(cgGetArrayParameter(param, i));
      }
      break;

      default: // Display parameter information
      const char *name = cgGetParameterName(param);
      CGtype paramType = cgGetParameterType(param);
      const char *type = cgGetTypeString(paramType);
      CGresource paramRes = cgGetParameterResource(param);
      const char *resource = cgGetResourceString(paramRes);

      printf("-- Name = '%s'\n", name);
      printf("-- Type = '%s'\n", type);
      printf("-- Resource = '%s'\n\n", resource);
    }
  } while((param = cgGetNextParameter(param)) != 0);
}
```

Setting Program Parameters

The **cgGLBindUniform()** and **cgGLBindVarying()** functions have been consolidated into the **cgGLSetParameter()** functions. The only other difference in the new runtime is that you no longer need to specify the program associated with a given parameter (this also applies to other parameter setting functions such as **cgGLSetStateMatrixParameter()** or **cgGLSetParameterPointer()**).

Example:

```
cgGLSetParameter4f(constantsParam, fSinTime * 13.0,
                   showNormals, 0.8, 1.0);
```

Setting Texture Parameters

The **cgGLActiveTexture()** and **cgGLClientActiveTexture()** functions have been removed and replaced with **cgGLSetTextureParameter()** and **cgGL{Enable/Disable}TextureParameter()** functions.

cgGLSetTextureParameter() is used to store the OpenGL texture object associated with the specified texture parameter. You then call **cgGLEnableTextureParameter()**, which will enable the texture unit associated with the texture parameter and bind the texture object set with **cgGLSetTextureParameter()**. **cgGLDisableTextureParameter()** will disable texturing for the texture unit associated with the texture parameter.

Example:

```
DiffuseTexture = cgGetNamedParameter(fragmentProgram,
                                     "diffuseMap");
cgGLSetTextureParameter(DiffuseTexture, texobj);

cgGLEnableTextureParameter(DiffuseTexture);

// ... Now draw geometry ...

cgGLDisableTextureParameter(DiffuseTexture);
```

Setting Parameter Arrays

You can now set an array of parameters using the **cgGLSetParameterArray()** functions. This allows you to declare an array of program parameters in your Cg shader (e.g. **uniform float4 pg[16]**), retrieve a single parameter handle for the array, and fill the array—all with a single call.

Example:

```
vec4f *pg = new vec4f[16];

pgArray = cgGetNamedParameter(program, "pg");
cgGLSetParameterArray4f(pgArray, 0, 16, pg[0].v);
```

To retrieve an individual parameter in the parameter array you can use the **cgGetArrayParameter()** function.

Example:

```
int index = 14;
CGparameter currentParam = cgGetArrayParameter(pgArray,
                                               index);
```

Setting State Matrices

To set state matrices you use the **cgGLSetStateMatrixParameter()** function instead of the **cgGLBindUniformStateMatrix()** function.

Example:

```
cgGLSetStateMatrixParameter(
                    modelViewProjParam,
                    CG_GL_MODELVIEW_PROJECTION_MATRIX,
                    CG_GL_MATRIX_IDENTITY);
```

Note: This function essentially reads the current matrix at the time its called and sets the associated parameter to the contents of the matrix. It does not perform any sort of state tracking.

Table 3 lists the old Cg matrix and Table 4 lists the matrix transformation types and their equivalents in the new runtime.

Table 3. Matrix types

Deprecated API	New API
cgGLModelViewProjectionMatrix	CG_GL_MODELVIEW_PROJECTION_MATRIX
cgGLModelViewMatrix	CG_GL_MODELVIEW_MATRIX
cgGLTextureMatrix	CG_GL_TEXTURE_MATRIX
cgGLProjectionMatrix	CG_GL_PROJECTION_MATRIX

Table 4. Matrix transformation types

Deprecated API	New API
cgGLMatrixIdentity	CG_GL_MATRIX_IDENTITY
cgGLMatrixInverse	CG_GL_MATRIX_INVERSE
cgGLMatrixTranspose	CG_GL_MATRIX_TRANSPOSE
cgGLMatrixInverse | cgGLMatrixTranspose	CG_GL_MATRIX_INVERSE_TRANSPOSE

Setting Vertex Arrays

The **cgGLBindVaryingPointer()** function has been replaced with the **cgGLSetParameterPointer()** function and the **cgGLEnableClientState()** function remains. The parameters are almost identical but the first parameter (the program associated with the parameter) has been removed.

Example:

```
Vertices = cgGetNamedParameter(vertexProgram,
                               "IN.Position");

cgGLEnableProfile(CG_PROFILE_VP20);
cgGLBindProgram(program);

cgGLEnableClientState(Vertices);
cgGLSetParameterPointer(Vertices, 3, GL_FLOAT,
                        0, vertexdata);

glDrawElements(...);

cgGLDisableClientState(Vertices);
```

You must bind your Cg program before calling **cgGLSetParameterPointer()**.

Enabling/Disabling a Profile

The **cgGLEnableProgramType()** and **cgGLDisableProfileType()** functions have been renamed to **cgGLEnableProfile()** and **cgGLDisableProfile()** respectively. Both functions take a **CGprofile** parameter specifying which profile to enable or disable.

Example:

```
cgGLEnableProfile(CG_PROFILE_VP20);

// ... draw geometry

cgGLDisableProfile(CG_PROFILE_VP20);
```

Binding a Program

To activate a specific program, call the **cgGLBindProgram()** function and specify which program you want bound. **cgGLBindProgram()** is the same as it was in the old runtime API.

Example:

```
cgGLEnableProfile(CG_PROFILE_VP20);
cgGLBindProgram(program);
```

When **cgGLBindProgram()** is called, it will reload any constant registers for the specified program with whatever value they were last set to. These constant registers correspond to uniform parameters, set using **cgGLSetParameter()**. This means that, if you wish, you can make these uniform **cgGLSetParameter()** calls before you call **cgGLBindProgram()**. The runtime will then store a copy of your data, and load it only when **cgGLBindProgram()** is finally called.

Keep in mind that this applies only to uniform parameters. Varying parameters such as vertex array pointers will not be reloaded. For example, you cannot call **cgGLSetParameterPointer()** just once—you would have to call it after each call to **cgGLBindProgram()**.

Resource Cleanup

The **cgFreeContext()** and **cgFreeProgramIter()** functions have been replaced with the **cgDestroyContext()** and **cgDestroyProgram()** functions. The **cgFreeBindIter()** function has been removed completely as there is no need to delete individual **CGparameter** handles, the runtime will clean these up for you when **cgDestroyProgram()** is called.

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