# A Follow-up Cg Runtime Tutorial for Readers of *The Cg Tutorial*[*]

***Mark J. Kilgard***
**NVIDIA Corporation**
**Austin, Texas**

**April 20, 2005**

*The source code discussed in this tutorial is installed by the Cg Toolkit installer for Windows at*
***`c:\Program Files\NVIDIA Corporation\Cg\examples\OpenGL\basic\24_bump_map_torus`***

When Randy and I wrote *The Cg Tutorial*,[†] we wanted a book that would convey our intense enthusiasm for programmable graphics using Cg,[‡] short for C for Graphics. We focused our tutorial on the language itself: What is the Cg language and how do you write Cg programs for programmable graphics hardware?

We chose our language focus for a couple of different reasons.

First off, the language is where all the power and new concepts are. Once you interface Cg into your graphics application, it's the Cg language that really matters. For a conventional CPU programming language, explaining the Cg runtime is somewhat akin to explaining how to edit programs and how to run the compiler. Obviously, you've got to learn these tasks, but there's nothing profound about using an editor or compiler. Likewise, there's nothing deep about the Cg runtime either; it's a fairly straightforward programming interface.

Second, how you interface Cg to your application is a matter of personal design and depends on the nature of your application and your choice of application programming language, operating system, and 3D programming interface. While Randy and I are happy to explain Cg and show how to program your graphics hardware with it, you are the person best able to interface Cg into your application code.

[*] You have permission to redistribute or make digital or hard copy of this article for non-commercial or educational use.

[†] *The Cg Tutorial* by Randima (Randy) Fernando and Mark J. Kilgard is published by Addison-Wesley (ISBN 0321194969, 336 pages). The book is now available in Japanese translation (ISBN4-939007-55-3).

[‡] *Cg in Two Pages* (http://xxx.lanl.gov/ftp/cs/papers/0302/0302013.pdf) by Mark J. Kilgard is a short overview of the Cg language. *Cg: A System for Programming Graphics Hardware in a C-like Language* (http://www.cs.utexas.edu/users/billmark/papers/Cg) by Bill Mark, Steve Glanville, Kurt Akeley, and Mark J. Kilgard is a SIGGRAPH 2003 paper explaining Cg's design in 12 pages.

Third, the language shares its design, syntax, and semantics with Microsoft's DirectX 9 High-Level Shader Language (HLSL). This means you can chose whether to use Microsoft's HLSL runtime (ideal for developers focused on DirectX for the Windows platform) or the Cg runtime—supplied by NVIDIA—for those of you who want to support a broad range of operating systems and 3D programming interfaces (such as Linux, Apple's OS X, and OpenGL). Because *The Cg Tutorial* focuses on the Cg language, all the concepts and syntax explained in the book apply whether you choose to use the Cg or HLSL implementation when it comes time to actually write your shader programs. Since there's been some confusion about this point, understand that *The Cg Tutorial* examples in the book compile with *either* language implementation. We hope *The Cg Tutorial* is an instructive book about both Cg *and* HLSL.

To avoid all the mundane details necessary to interface Cg programs to a real application, *The Cg Tutorial* includes an accompanying CD-ROM* with a software framework so you can examine and modify the various Cg programs in the book and see the rendering results without worrying about the mundane details of writing a full application, loading models and textures, and interfacing Cg to your application. Still, the book does provide a brief appendix describing the Cg runtime programming interface for both OpenGL and Direct3D.

## 1. Follow-up: A Complete Cg Demo

Still, there's not a *complete* basic example that shows how everything fits together. With that in mind, this article presents a complete graphics demo written in ANSI C that renders a procedurally-generated bump-mapped torus. The demo's two Cg programs are taken directly from the book's Chapter 8 (Bump Mapping). While the Cg programs are reprinted at the end of the article, please consult *The Cg Tutorial* for an explanation of the programs and the underlying bump mapping background and mathematics.

The demo renders with OpenGL and interfaces with the window system via the cross-platform OpenGL Utility Toolkit (GLUT).† To interface the application with the Cg programs, the demo calls the generic Cg and OpenGL-specific CgGL runtime routines.

OpenGL, GLUT, and the Cg and CgGL runtimes are supported on Windows, OS X, and Linux so the demo source code compiles and runs on all these operating systems. The demo automatically selects the most appropriate profile for your hardware. Cg supports multi-vendor OpenGL profiles (namely, **`arbvp1`** and **`arbfp1`**) so the demo works on

* You can download the latest version of the software accompanying *The Cg Tutorial* from http://developer.nvidia.com/object/cg_tutorial_software.html for either Windows or Linux. For best results, make sure you have the latest graphics drivers, latest Cg toolkit, and latest version of *The Cg Tutorial* examples installed.

† Documentation, source code, and pre-compiled GLUT libraries are available from http://www.opengl.org/developers/documentation/glut.html

GPUs from ATI, NVIDIA, or any other OpenGL implementation, such as Brian Paul's open source Mesa library, that exposes the multi-vendor **`ARB_vertex_program`** and **`ARB_fragment_program`** OpenGL extensions.

I verified the demo works on DirectX 9-class hardware including ATI's Radeon 9700 and similar GPUs, NVIDIA's GeForce FX products, and the GeForce 6 Series. The demo even works on older NVIDIA DirectX 8-class hardware such as GeForce3 and GeForce4 Ti GPUs.

So this article's simple Cg-based demo handles multiple operating systems, two different GPU hardware generations (DirectX 8 & DirectX 9), and hardware from the two major GPU vendors (and presumably any other OpenGL implementation exposing OpenGL's standard, multi-vendor vertex and fragment program extensions) with absolutely no GPU-dependent or operating system-dependent code.

To further demonstrate the portability possible by writing shaders in Cg, you can also compile the discussed Cg programs with Microsoft's HLSL runtime with no changes to the Cg programs.

This unmatched level of shader portability is why the Cg language radically changes how graphics applications get at programmable shading hardware today. With one high-level language, you can write high-performance, cross-platform, cross-vendor, and cross-3D API shaders. Just as you can interchange images and textures stored as JPEG, PNG, and Targa files across platforms, you can now achieve a similar level of interoperability with something as seemingly hardware-dependent as a hardware shading algorithm.

## 2. Demo Source Code Walkthrough

The demo, named **`cg_bumpdemo`**, consists of the following five source files:

1. **`cg_bumpdemo.c`**—ANSI C source code for the demo.
2. **`brick_image.h`**—Header file containing RGB8 image data for a mipmapped 128x128 normal map for a brick pattern.
3. **`nmap_image.h`**—Header file containing RGB8 image data for a normalization vector cube map with 32x32 faces.
4. **`C8E6v_torus.cg`**—Cg vertex program to generate a torus from a 2D mesh of vertices.
5. **`C8E4f_specSurf.cg`**—Cg fragment program for surface-local specular and diffuse bump mapping.

Later, we will go through **`cg_bumpdemo.c`** line-by-line.

To keep the demo self-contained and maintain the focus on how the Cg runtime loads, compiles, and configures the Cg programs and then renders with them, this demo uses static texture image data included in the two header files.

The data in these header files are used to construct OpenGL texture objects for a brick pattern normal map 2D texture and a "vector normalization" cube map. These texture objects are sampled by the fragment program.

The data in the two headers files consists of hundreds of comma-separated numbers (I'll save you the tedium of publishing all the numbers in this article…). Rather than static data compiled into an executable, a typical application would read normal map textures from on-disk image files or convert a height-field image file to a normal map. Likewise, a "normalization vector" cube map is typically procedurally generated rather than loaded from static data.

The two Cg files each contain a Cg entry function with the same name as the file. These functions are explained in Chapter 8 (Bump Mapping) of *The Cg Tutorial*. These files are read by the demo when the demo begins running. The demo uses the Cg runtime to read, compile, configure, and render with these Cg programs.

Rather than rehash the background, theory, and operation of these Cg programs, you should consult Chapter 8 of *The Cg Tutorial*. Pages 200 to 204 explain the construction of the brick pattern normal map. Pages 206 to 208 explain the construction and application of a normalization cube map. Pages 208 to 211 explains specular bump mapping, including the **`C8E4f_specSurf`** fragment program. Pages 211 to 218 explain texture-space bump mapping. Pages 218 to 224 explain the construction of the per-vertex coordinate system needed for texture-space bump mapping for the special case of an object (the torus) that is generated from parametric equations by the **`C8E6v_torus`** vertex program.

For your convenience and so you can map Cg parameter names used in the C source file to their usage in the respective Cg programs, the complete contents of **`C8E6v_torus.cg`** and **`C8E4f_specSurf.cg`** are presented in Appendix A and Appendix B at the end of this article (the Cg programs are short, so why not).

## 3. On to the C Code

Now, it's time to dissect **`cg_bumpdemo.c`** line-by-line as promised (we'll skip comments in the source code if the comments are redundant with the discussion below).

To help you identify which names are external to the program, the following words are listed in **`boldface`** within the C code: C keywords; C standard library routines and macros; OpenGL, GLU, and GLUT routines, types, and enumerants; and Cg and CgGL runtime routines, types, and enumerants.

## 3.1. *Initial Declarations*

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
#include <Cg/cg.h>
#include <Cg/cgGL.h>
```

The first three includes are basic ANSI C standard library includes. We'll be using **`sin`**, **`cos`**, **`printf`**, **`exit`**, and **`NULL`**. We rely on the GLUT header file to include the necessary OpenGL and OpenGL Utility Library (GLU) headers.

The **`<Cg/cg.h>`** header contains generic routines for loading and compiling Cg programs but does not contain routines that call the 3D programming interface to configure the Cg programs for rendering. The generic Cg routines begin with a **`cg`** prefix; the generic Cg types begin with a **`CG`** prefix; and the generic Cg macros and enumerations begin with a **`CG_`** prefix.

The **`<Cg/cgGL.h>`** contains the OpenGL-specific routines for configuring Cg programs for rendering with OpenGL. The OpenGL-specific Cg routines begin with a **`cgGL`** prefix; the OpenGL-specific Cg types begin with a **`CGGL`** prefix; and the OpenGL-specific Cg macros and enumerations begin with a **`CGGL_`** prefix.

Technically, the **`<Cg/cgGL.h>`** header includes **`<Cg/cg.h>`** so we don't have to explicitly include **`<Cg/cg.h>`** but we include both to remind you that we'll be calling both generic Cg routines and OpenGL-specific Cg routines.

```
/* An OpenGL 1.2 define */
#define GL_CLAMP_TO_EDGE                    0x812F

/* A few OpenGL 1.3 defines */
#define GL_TEXTURE_CUBE_MAP                 0x8513
#define GL_TEXTURE_BINDING_CUBE_MAP         0x8514
#define GL_TEXTURE_CUBE_MAP_POSITIVE_X      0x8515
```

We will use these OpenGL enumerants later when initializing our "normalization vector" cube map. We list them here explicitly since we can't count on **`<GL/gl.h>`** (included by **`<GL/glut.h>`** above) to have enumerants added since OpenGL 1.1 because Microsoft still supplies the dated OpenGL 1.1 header file.

Next, we'll list all global variables we plan to use. We use the **`my`** prefix to indicate global variables that we define (to make it crystal clear what names we are defining rather than those names defined by header files). When we declare a variable of a type defined by the Cg runtime, we use the **`myCg`** prefix to remind you that the variable is for use with the Cg runtime.

### 3.1.1. Cg Runtime Variables

```
static CGcontext   myCgContext;
static CGprofile   myCgVertexProfile,
                   myCgFragmentProfile;
static CGprogram   myCgVertexProgram,
                   myCgFragmentProgram;
static CGparameter myCgVertexParam_lightPosition,
                   myCgVertexParam_eyePosition,
                   myCgVertexParam_modelViewProj,
                   myCgVertexParam_torusInfo,
                   myCgFragmentParam_ambient,
                   myCgFragmentParam_LMd,
                   myCgFragmentParam_LMs,
                   myCgFragmentParam_normalMap,
                   myCgFragmentParam_normalizeCube,
                   myCgFragmentParam_normalizeCube2;
```

These are the global Cg runtime variables the demo initializes uses. We need a single Cg compilation context named **`myCgContext`**. Think of your Cg compilation context as the "container" for all the Cg handles you manipulate. Typically your program requires just one Cg compilation context.

We need two Cg profile variables, one for our vertex program profile named **`myCgVertexProfile`** and another for our fragment program profile named **`myCgFragmentProfile`**. These profiles correspond to a set of programmable hardware capabilities for vertex or fragment processing and their associated execution environment. Profiles supported by newer GPUs are generally more functional than older profiles. The Cg runtime makes it easy to select the most appropriate profile for your hardware as we'll see when we initialize these profile variables.

Next we need two Cg program handles, one for our vertex program named **`myCgVertexProgram`** and another for our fragment program named **`myCgFragmentProgram`**. When we compile a Cg program successfully, we use these handles to refer to the corresponding compiled program.

We'll need handles to each of the uniform input parameters used by our vertex and fragment programs respectively. We use these handles to match the uniform input parameters in the Cg program text with the opaque OpenGL state used to maintain the corresponding Cg program state. Different profiles can maintain Cg program state with different OpenGL state so these Cg parameter handles abstract away the details of how a particular profile manages a particular Cg parameter.

The **`myCgVertexParam_`** prefixed parameter handles end with each of the four uniform input parameters to the **`C8E6v_torus`** vertex program in Appendix A. Likewise, the **`myCgFragmentParam_`** prefixed parameter handles end with each of the six uniform input parameters to the **`C8E4v_specSurf`** fragment program in Appendix B.

In a real program, you'll probably have more Cg program handles than just two. You may have hundreds depending on how complicated the shading is in your application. And each program handle requires a Cg parameter handle for each input parameter. This means you probably won't want to use global variables to store these handles. You'll probably want to encapsulate your Cg runtime handles within "shader objects" that may well combine vertex and fragment Cg programs and their parameters within the same object for convenience. Keep in mind that this demo is trying to be very simple.

### 3.1.2. Other Variables

```
static const char *myProgramName = "cg_bumpdemo",
                  *myVertexProgramFileName = "C8E6v_torus.cg",
                  *myVertexProgramName = "C8E6v_torus",
                  *myFragmentProgramFileName = "C8E4f_specSurf.cg",
                  *myFragmentProgramName = "C8E4f_specSurf";
```

We need various string constants to identify our program name (for error messages and the window name), the names of the file names containing the text of the vertex and fragment Cg programs to load, and the names of the entry functions for each of these files.

In Appendix A, you'll find the contents of the **`C8E6v_torus.cg`** file and, within the file's program text, you can find the entry function named **`C8E6v_torus`**. In Appendix B, you'll find the contents of the **`C8E4f_specSurf.cg`** file and, within the file's program text, you can find the entry function name **`C8E4f_specSurf`**.

```
static float myEyeAngle = 0,
             myAmbient[4] = { 0.3f, 0.3f, 0.3f, 0.3f }, /* Dull white */
             myLMd[4] = { 0.9f, 0.6f, 0.3f, 1.0f },     /* Gold */
             myLMs[4] = { 1.0f, 1.0f, 1.0f, 1.0f };     /* Bright white */
```

These are demo variables used to control the rendering of the scene. The viewer rotates around the fixed torus. The angle of rotation and a degree of elevation for the viewer is determined by **`myEyeAngle`**, specified in radians. The other three variables provide lighting and material parameters to the fragment program parameters. With these particular values, the bump-mapped torus has a "golden brick" look.

### 3.1.3. Texture Data

```
/* OpenGL texture object (TO) handles. */
enum {
  TO_NORMALIZE_VECTOR_CUBE_MAP = 1,
  TO_NORMAL_MAP = 2,
};
```

The **`TO_`** prefixed enumerants provide numbers for use as OpenGL texture object names.

```
static const GLubyte
myBrickNormalMapImage[3*(128*128+64*64+32*32+16*16+8*8+4*4+2*2+1*1)] = {
/* RGB8 image data for mipmapped 128x128 normal map for a brick pattern */
#include "brick_image.h"
};

static const GLubyte
myNormalizeVectorCubeMapImage[6*3*32*32] = {
/* RGB8 image data for normalization vector cube map with 32x32 faces */
#include "normcm_image.h"
};
```

These static, constant arrays include the header files containing the data for the normal map's brick pattern and the "normalization vector" cube map. Each texel is 3 unsigned bytes (one for red, green, and blue). While each byte of the texel format is unsigned, normal map components, as well as the vector result of normalizing an arbitrary direction vector, are logically signed values within the [-1,1] range. To accommodate signed values with OpenGL's conventional **`GL_RGB8`** unsigned texture format, the unsigned [0,1] range is expanded in the fragment program to a signed [-1,1] range. This is the reason for the **`expand`** helper function called by the **`C8E4f_specSurf`** fragment program (see Appendix B).

The normal map has mipmaps so there is data for the 128x128 level, and then, each of the successively downsampled mipmap levels. The "normalization vector" cube map has six 32x32 faces.

## *3.2. Error Reporting Helper Routine*

```
static void checkForCgError(const char *situation)
{
  CGerror error;
  const char *string = cgGetLastErrorString(&error);

  if (error != CG_NO_ERROR) {
    printf("%s: %s: %s\n",
      myProgramName, situation, string);
    if (error == CG_COMPILER_ERROR) {
      printf("%s\n", cgGetLastListing(myCgContext));
    }
    exit(1);
  }
}
```

Cg runtime routines report errors by setting a global error value. Calling the **`cgGetLastErrorString`** routine both returns a human-readable string describing the last generated Cg error and writes an error code of type **`CGerror`**. **`CG_NO_ERROR`** (defined to be zero) means there was no error. As a side-effect, **`cgGetLastErrorString`** also resets the global error value to **`CG_NO_ERROR`**. The Cg runtime also includes the simpler function **`cgGetError`** that just returns and then resets the global error code if you just want the error code and don't need a human-readable string too.

The **checkForCgError** routine is used to ensure proper error checking throughout the demo. Rather than cheap out on error checking, the demo checks for errors after essentially every Cg runtime call by calling **checkForCgError**. If an error has occurred, the routine prints an error message including the **situation** string and translated Cg error value string, and then exits the demo.

When the error returned is **CG_COMPILER_ERROR** that means there are compiler error messages too. So **checkForCgError** then calls **cgGetLastListing** to get a listing of the compiler error messages and prints these out too. For example, if your Cg program had a syntax error, you'd see the compiler's error messages including the line numbers where the compiler identified problems.

While "just exiting" is fine for a demo, real applications will want to properly handle any errors generated. In general, you don't have to be so paranoid as to call **cgGetLastErrorString** after every Cg runtime routine. Check the runtime API documentation for each routine for the reasons it can fail; when in doubt, check for failures.

## *3.3. Demo Initialization*

```
static void display(void);
static void keyboard(unsigned char c, int x, int y);

int main(int argc, char **argv)
{
  const GLubyte *image;
  unsigned int size, level, face;
```

The **main** entry-point for the demo needs a few local variables to be used when loading textures. We also need to forward declare the **display** and **keyboard** GLUT callback routines for redrawing the demo's rendering window and handling keyboard events.

### 3.3.1. OpenGL Utility Toolkit Initialization

```
  glutInitWindowSize(400, 400);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutInit(&argc, argv);

  glutCreateWindow(myProgramName);
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
```

Using GLUT, we request a double-buffered RGB color 400x400 window with a depth buffer. We allow GLUT to take a pass parsing the program's command line arguments. Then, we create a window and register the **display** and **keyboard** callbacks. We'll explain these callback routines after completely initializing GLUT, OpenGL, and Cg. That's it for initializing GLUT except for calling **glutMainLoop** to start event processing at the very end of **main**.

### 3.3.2. OpenGL Rendering State Initialization

```
glClearColor(0.1, 0.3, 0.6, 0.0);  /* Blue background */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(
  60.0,    /* Field of view in degree */
  1.0,     /* Aspect ratio */
  0.1,     /* Z near */
  100.0); /* Z far */
glMatrixMode(GL_MODELVIEW);
glEnable(GL_DEPTH_TEST);
```

Next, we initialize basic OpenGL rendering state. For better aesthetics, we change the background color to a nice sky blue. We specify a perspective projection matrix and enable depth testing for hidden surface elimination.

### 3.3.3. OpenGL Texture Object Initialization

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); /* Tightly packed texture data. */
```

By default, OpenGL's assumes each image scanline is aligned to begin on 4 byte boundaries. However, RGB8 data (3 bytes per pixel) is usually tightly packed so a 1 byte alignment is appropriate. That's indeed the case for the RGB8 pixels in our static arrays used to initialize our textures. If you didn't know about this OpenGL pitfall before, you do now.[‡]

#### 3.3.3.1. Normal Map 2D Texture Initialization

```
glBindTexture(GL_TEXTURE_2D, TO_NORMAL_MAP);
/* Load each mipmap level of range-compressed 128x128 brick normal
   map texture. */
for (size = 128, level = 0, image = myBrickNormalMapImage;
     size > 0;
     image += 3*size*size, size /= 2, level++) {
  glTexImage2D(GL_TEXTURE_2D, level,
    GL_RGB8, size, size, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_LINEAR);
```

We bind to the texture object for our brick pattern normal map 2D texture and load each of the 7 mipmap levels, starting with the 128x128 base level and working down to the 1x1 level. Each level is packed into the **`myBrickNormalMapImage`** array right after the

[‡] Being aware of pitfalls such as this one can save you a lot of time debugging. This and other OpenGL pitfalls are enumerated in my article "Avoiding 19 Common OpenGL Pitfalls" found here http://developer.nvidia.com/object/Avoiding_Common_ogl_Pitfalls.html An earlier HTML version of the article (with just 16 pitfalls) is found here http://www.opengl.org/developers/code/features/KilgardTechniques/oglpitfall/oglpitfall.html

previous level. So the 64x64 mipmap level immediately follows the 128x128 level, and so on. OpenGL's default minification filter is "nearest mipmap linear" (again, a weird default—it means nearest filtering within a mipmap level and then bilinear filtering between the adjacent mipmap levels) so we switch to higher-quality "linear mipmap linear" filtering.

#### 3.3.3.2. Normalize Vector Cube Map Texture Initialization

```
glBindTexture(GL_TEXTURE_CUBE_MAP, TO_NORMALIZE_VECTOR_CUBE_MAP);
/* Load each 32x32 face (without mipmaps) of range-compressed "normalize
   vector" cube map. */
for (face = 0, image = myNormalizeVectorCubeMapImage;
     face < 6;
     face++, image += 3*32*32) {
  glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + face, 0,
    GL_RGB8, 32, 32, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
}
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
  GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
  GL_CLAMP_TO_EDGE);
```

Next, we bind the texture object for the "normalization vector" cube map[1] intended to quickly normalize the 3D lighting vectors that are passed as texture coordinates. The cube map texture has six faces but there's no need for mipmaps. Each face is packed into the **`myNormalizeVectorCubeMapImage`** array right after the prior face with the faces ordered in the order of the sequential texture cube map face OpenGL enumerants.

Again, the default minification state is inappropriate (this time because we don't have mipmaps) so **`GL_LINEAR`** is specified instead. While the default **`GL_REPEAT`** wrap mode was fine for the brick pattern that we intend to tile over the surface of the torus, the **`GL_CLAMP_TO_EDGE`** wrap mode (introduced by OpenGL 1.2) keeps one edge of a cube map face from bleeding over to the other.

GLUT and OpenGL are now initialized so it is time to begin loading, compiling, and configuring the Cg programs.

### 3.3.4. Cg Runtime Initialization

```
myCgContext = cgCreateContext();
checkForCgError("creating context");
```

[1] Using a "normalization vector" cube map allows our demo to work on older DirectX 8-class GPUs that lacked the shading generality to normalize vectors mathematically. Ultimately as more capable GPUs become ubiquitous, use of normalization cube maps is sure to disappear in favor of normalizing a vector mathematically. See Exercise 5.

Before we can do anything with the Cg runtime, we need to allocate a Cg compilation context with **`cgCreateContext`**. Typically, your application just needs one Cg compilation context unless you have a multi-threaded application that requires using the Cg runtime concurrently in different threads. Think of the Cg context as the context and container for all your Cg programs that are creating, loading (compiling), and configured by the Cg runtime.

### 3.3.5. Cg Vertex Profile Selection

```
myCgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
cgGLSetOptimalOptions(myCgVertexProfile);
checkForCgError("selecting vertex profile");
```

We need a profile with which to compile our vertex program. We could hard-code a particular profile (for example, the multi-vendor **`CG_PROFILE_ARBVP1`** profile), but we are better off asking the CgGL runtime to determine the best vertex profile for our current OpenGL context by calling the **`cgGLGetLatestProfile`** routine. (Keep in mind there's a current OpenGL rendering context that GLUT created for us when we called **`glutCreateWindow`**.) **`cgGLGetLatestProfile`** calls OpenGL queries to examine the current OpenGL rendering context. Based on the OpenGL **`GL_EXTENSIONS`** string, this routine can decide what profiles are supported and then which hardware-supported profile offers the most functionality and performance. The **`CG_GL_VERTEX`** parameter says to return the most appropriate vertex profile, but we can also pass **`CG_GL_FRAGMENT`**, as we will do later, to determine the most appropriate fragment profile.

Cg supports a number of vertex profiles. These are the vertex profiles currently supported by Cg 1.4 for OpenGL: **`CG_PROFILE_VP40`** corresponds to the **`vp40`** vertex program profile for the **`NV_vertex_program3`** OpenGL extension (providing full access to the vertex processing features of NVIDIA's GeForce 6 Series GPUs such as vertex textures). **`CG_PROFILE_VP30`** corresponds to the **`vp30`** vertex program profile for the **`NV_vertex_program2`** OpenGL extension (providing full access to the vertex processing features of NVIDIA's GeForce FX GPUs such as per-vertex dynamic branching). **`CG_PROFILE_ARBVP1`** corresponds to the **`arbvp1`** vertex program profile for the **`ARB_vertex_program`** OpenGL extension (a multi-vendor OpenGL standard, supported by both NVIDIA and ATI). **`CG_PROFILE_VP20`** corresponds to the **`vp20`** vertex program profile for the **`NV_vertex_program`** and **`NV_vertex_program1_1`** OpenGL extensions (for NVIDIA's GeForce3, GeForce4 Ti, and later GPUs).

While several GPUs can support the same profile, there may be GPU-specific techniques the Cg compiler can use to make the most of the available functionality and generate better code for your given GPU. By calling **`cgGLSetOptimalOptions`** with the profile we've selected, we ask the compiler to optimize for the specific hardware underlying our OpenGL rendering context.

For example, some vertex profiles such as **`CG_PROFILE_VP40`** support texture fetches but typically support fewer texture image units than the hardware's corresponding fragment-level texture functionality. **`cgGLSetOptimalOptions`** informs the compiler what the hardware's actual vertex texture image unit limit is.

### 3.3.5.1. Vertex Program Creation and Loading

```
myCgVertexProgram =
  cgCreateProgramFromFile(
    myCgContext,              /* Cg runtime context */
    CG_SOURCE,                /* Program in human-readable form */
    myVertexProgramFileName,  /* Name of file containing program */
    myCgVertexProfile,        /* Profile to try */
    myVertexProgramName,      /* Entry function name */
    NULL);                    /* No extra compiler options */
checkForCgError("creating vertex program from file");
cgGLLoadProgram(myCgVertexProgram);
checkForCgError("loading vertex program");
```

Now we try to create and load the Cg vertex program. We use the optimal vertex profile for our OpenGL rendering context to compile the vertex program contained in the file named by **`myVertexProgramFileName`**. As it turns out, the **`C8E6v_torus`** vertex program is simple enough that every Cg vertex profile mentioned in the last section is functional enough to compile the **`C8E6v_torus`** program.

The **`cgCreateProgramFromFile`** call reads the file, parses the contents, and searches for the entry function specified by **`myVertexProgramName`** and, if found, creates a vertex program for the profile specified by **`myCgVertexProfile`**. The **`cgCreateProgramFromFile`** routine is a generic Cg runtime routine so it just creates the program without actually translating the program into a form that can be passed to the 3D rendering programming interface.

You don't actually need a current OpenGL rendering context to call **`cgCreateProgramFromFile`**, but you do need a current OpenGL rendering context that supports the profile of the program for **`cgGLLoadProgram`** to succeed.

It is the OpenGL-specific **`cgGLLoadProgram`** routine that translates the program into a profile-dependent form. For example, in the case of the multi-vendor **`arbvp1`** profile, this includes calling the **`ARB_vertex_program`** extension routine **`glProgramStringARB`** to create an OpenGL program object.

We expect **`cgGLLoadProgram`** to "just work" because we've already selected a profile suited for our GPU and **`cgCreateProgramFromFile`** successfully compiled the Cg program into a form suitable for that profile.

#### 3.3.5.2. Vertex Program Parameter Handles

```
myCgVertexParam_lightPosition =
  cgGetNamedParameter(myCgVertexProgram, "lightPosition");
checkForCgError("could not get lightPosition parameter");

myCgVertexParam_eyePosition =
  cgGetNamedParameter(myCgVertexProgram, "eyePosition");
checkForCgError("could not get eyePosition parameter");

myCgVertexParam_modelViewProj =
  cgGetNamedParameter(myCgVertexProgram, "modelViewProj");
checkForCgError("could not get modelViewProj parameter");

myCgVertexParam_torusInfo =
  cgGetNamedParameter(myCgVertexProgram, "torusInfo");
checkForCgError("could not get torusInfo parameter");
```

Now that the vertex program is created and successfully loaded, we initialize all the Cg parameter handles. Later during rendering in the **`display`** callback, we will use these parameter handles to update whatever OpenGL state the compiled program associates with each parameter.

In this demo, we know *a priori* what the input parameter names are to keep things simple. If we had no special knowledge of the parameter names, we could use Cg runtime routines to iterate over all the parameter names for a given program (see the **`cgGetFirstParameter`**, **`cgGetNextParameter`**, and related routines—use these for Exercise 11 at the end of this article).

### 3.3.6. Cg Fragment Profile Selection

```
myCgFragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
cgGLSetOptimalOptions(myCgFragmentProfile);
checkForCgError("selecting fragment profile");
```

We select our fragment profile in the same manner we used to select our vertex profile. The only difference is we pass the **`CG_GL_FRAGMENT`** parameter when calling **`cgGLGetLatestProfile`**.

Cg supports a number of fragment profiles. These are the fragment profiles currently supported by Cg 1.4 for OpenGL: **`CG_PROFILE_FP40`** corresponds to the **`fp40`** vertex program profile for the **`NV_fragment_program2`** OpenGL extension (providing full access to the fragment processing features of NVIDIA's GeForce 6 Series GPUs such as per-fragment dynamic branching). **`CG_PROFILE_FP30`** corresponds to the **`fp30`** vertex program profile for the **`NV_fragment_program`** OpenGL extension (providing full access to the fragment processing features of NVIDIA's GeForce FX GPUs). **`CG_PROFILE_ARBFP1`** corresponds to the **`arbfp1`** fragment program profile for the **`ARB_fragment_program`** OpenGL extension (a multi-vendor OpenGL standard, supported by both NVIDIA and ATI). **`CG_PROFILE_FP20`** corresponds to the **`fp20`** vertex program profile for the **`NV_texture_shader`**, **`NV_texture_shader2`**,

**NV_register_combiners,** and **NV_register_combiners2** OpenGL extensions (for NVIDIA's GeForce3, GeForce4 Ti, and later GPUs).

As in the vertex profile case, **cgGLSetOptimalOptions** informs the compiler about specific hardware limits relevant to fragment profiles. For example, when your OpenGL implementation supports the **ATI_draw_buffers** extension, the **cgGLSetOptimalOptions** informs the compiler of this fact so the compiler can know how many color buffers are actually available when compiling for fragment profiles that support output multiple color buffers. Other limits such as the **ARB_fragment_program** limit on texture indirections are likewise queried so the compiler is aware of this limit. The maximum number of texture indirections the GPU can support may require the compiler to re-schedule the generated instructions around this limit. Other profile limits include the number of texture image units available, the maximum number of temporaries and constants allowed, and the static instruction limit.

#### 3.3.6.1. Fragment Program Creation and Loading

```
myCgFragmentProgram =
  cgCreateProgramFromFile(
    myCgContext,                /* Cg runtime context */
    CG_SOURCE,                  /* Program in human-readable form */
    myFragmentProgramFileName,  /* Name of file containing program */
    myCgFragmentProfile,        /* Profile to try */
    myFragmentProgramName,      /* Entry function name */
    NULL);                      /* No extra compiler options */
checkForCgError("creating fragment program from file");
cgGLLoadProgram(myCgFragmentProgram);
checkForCgError("loading fragment program");
```

We create and load the fragment program in much the same manner as the vertex program.

#### 3.3.6.2. Fragment Program Parameter Handles

```
myCgFragmentParam_ambient =
  cgGetNamedParameter(myCgFragmentProgram, "ambient");
checkForCgError("getting ambient parameter");

myCgFragmentParam_LMd =
  cgGetNamedParameter(myCgFragmentProgram, "LMd");
checkForCgError("getting LMd parameter");

myCgFragmentParam_LMs =
  cgGetNamedParameter(myCgFragmentProgram, "LMs");
checkForCgError("getting LMs parameter");

myCgFragmentParam_normalMap =
  cgGetNamedParameter(myCgFragmentProgram, "normalMap");
checkForCgError("getting normalMap parameter");

myCgFragmentParam_normalizeCube =
  cgGetNamedParameter(myCgFragmentProgram, "normalizeCube");
checkForCgError("getting normalizeCube parameter");
```

```
myCgFragmentParam_normalizeCube2 =
  cgGetNamedParameter(myCgFragmentProgram, "normalizeCube2");
checkForCgError("getting normalizeCube2 parameter");
```

We initialize input parameter handles in the same manner as done for vertex parameter handles.

#### 3.3.6.3. Setting OpenGL Texture Objects for Sampler Parameters

```
cgGLSetTextureParameter(myCgFragmentParam_normalMap,
  TO_NORMAL_MAP);
checkForCgError("setting normal map 2D texture");

cgGLSetTextureParameter(myCgFragmentParam_normalizeCube,
  TO_NORMALIZE_VECTOR_CUBE_MAP);
checkForCgError("setting 1st normalize vector cube map");

cgGLSetTextureParameter(myCgFragmentParam_normalizeCube2,
  TO_NORMALIZE_VECTOR_CUBE_MAP);
checkForCgError("setting 2nd normalize vector cube map");
```

Parameter handles for sampler parameters need to be associated with OpenGL texture objects. The first **`cgGLSetTextureParameter`** call associates the **`TO_NORMAL_MAP`** texture object with the **`myCgFragmentParam_normalMap`** parameter handle.

Notice how the **`TO_NORMALIZE_VECTOR_CUBE_MAP`** texture object is associated with the *two* distinct sampler parameters, **`normalizeCube`** and **`normalizeCube2`**. The reason this is done is to support older DirectX 8-class hardware such as the GeForce3 and GeForce4 Ti. These older DirectX 8-class GPUs must sample the texture associated with a given texture unit and that unit's corresponding texture coordinate set (and *only* that texture coordinate set). In order to support DirectX 8-class profiles (namely, **`fp20`**), the **`C8E4f_specSurf`** fragment program is written in such a way that the texture units associated with the two 3D vectors to be normalized (**`lightDirection`** and **`halfAngle`**) are each bound to the same "normalization vector" cube map. If there was no desire to support older DirectX 8-class hardware, fragment programs targeting the more general DirectX 9-class profiles (namely, **`arbfp1`** and **`fp30`**) could simply sample a single "normalization vector" texture unit.

Alternatively, the Cg fragment program could normalize the 3D lighting vectors with the **`normalize`** Cg standard library routine (see Exercise 5 at the end of this article), but for a lot of current hardware, a "normalization vector" cube map is faster and the extra precision for a mathematical normalize function is not crucial for lighting.

### 3.3.7. Start Event Processing

```
  glutMainLoop();
  return 0;  /* Avoid a compiler warning. */
}
```

GLUT, OpenGL, and Cg are all initialized now so we can start GLUT event processing. This routine never returns. When a redisplay of the GLUT window created earlier is needed, the **`display`** callback is called. When a key press occurs in the window, the **`keyboard`** callback is called.

## *3.4. Displaying the Window*

Earlier in the code, we forward declared the **`display`** callback. Now it's time to discuss what the **`display`** routine does and how exactly we render our bump-mapped torus using the textures and Cg vertex and fragment programs we've loaded.

### 3.4.1. Rendering a 2D Mesh to Generate a Torus

In the course of updating the window, the **`display`** callback invokes the **`drawFlatPatch`** subroutine. This subroutine renders a flat 2D mesh with immediate-mode OpenGL commands.

```
/* Draw a flat 2D patch that can be "rolled & bent" into a 3D torus by
   a vertex program. */
void
drawFlatPatch(float rows, float columns)
{
  const float m = 1.0f/columns;
  const float n = 1.0f/rows;
  int i, j;

  for (i=0; i<columns; i++) {
    glBegin(GL_QUAD_STRIP);
    for (j=0; j<=rows; j++) {
      glVertex2f(i*m, j*n);
      glVertex2f((i+1)*m, j*n);
    }
    glVertex2f(i*m, 0);
    glVertex2f((i+1)*m, 0);
    glEnd();
  }
}
```

The mesh consists of a number of adjacent quad strips. The **`C8E6v_torus`** vertex program will take these 2D vertex coordinates and use them as parametric coordinates for evaluating the position of vertices on a torus.

Nowadays it's much faster to use OpenGL vertex arrays, particularly with vertex buffer objects, to render geometry, but for this simple demo, immediate mode rendering is easier.

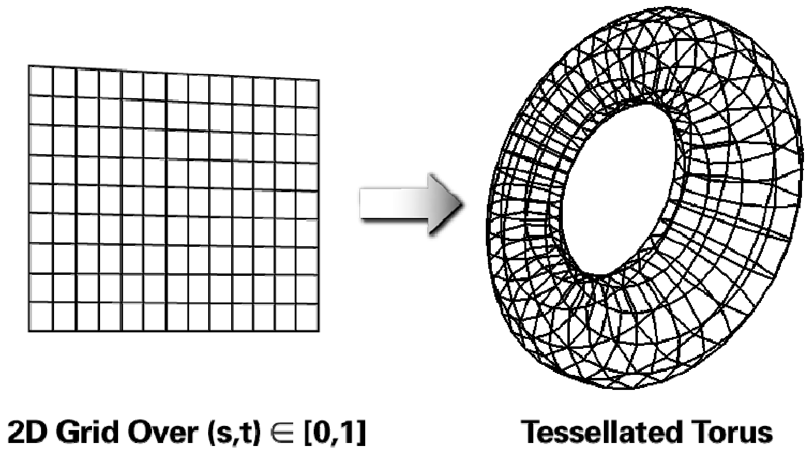


Figure 8-7 from *The Cg Tutorial* is replicated to illustrate how a 2D mesh could be procedurally "rolled and bent" into a torus by a vertex program.

### 3.4.2. The Display Callback

```
static void display(void)
{
  const float outerRadius = 6, innerRadius = 2;
  const int sides = 20, rings = 40;
  const float eyeRadius = 18.0;
  const float eyeElevationRange = 8.0;
  float eyePosition[3];

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

The **display** callback has a number of constants that control the torus size and tessellation and how the torus is viewed.

```
  eyePosition[0] = eyeRadius * sin(myEyeAngle);
  eyePosition[1] = eyeElevationRange * sin(myEyeAngle);
  eyePosition[2] = eyeRadius * cos(myEyeAngle);

  glLoadIdentity();
  gluLookAt(
    eyePosition[0], eyePosition[1], eyePosition[2],
    0.0 ,0.0,  0.0,   /* XYZ view center */
    0.0, 1.0,  0.0);  /* Up is in positive Y direction */
```

The viewing transform is re-specified each frame. The eye position is a function of **myEyeAngle**. By animating this variable, the viewer rotates around the torus with a sinusoidally varying elevation. Because specular bump mapping is view-dependent, the specular lighting varies over the torus as the viewer rotates around.

### 3.4.2.1. Binding, Configuring, and Enabling the Vertex Program

```
cgGLBindProgram(myCgVertexProgram);
checkForCgError("binding vertex program");

cgGLSetStateMatrixParameter(myCgVertexParam_modelViewProj,
                            CG_GL_MODELVIEW_PROJECTION_MATRIX,
                            CG_GL_MATRIX_IDENTITY);
checkForCgError("setting modelview-projection matrix");
cgGLSetParameter3f(myCgVertexParam_lightPosition, -8, 0, 15);
checkForCgError("setting light position");
cgGLSetParameter3fv(myCgVertexParam_eyePosition, eyePosition);
checkForCgError("setting eye position");
cgGLSetParameter2f(myCgVertexParam_torusInfo, outerRadius, innerRadius);
checkForCgError("setting torus information");

cgGLEnableProfile(myCgVertexProfile);
checkForCgError("enabling vertex profile");
```

Prior to rendering the 2D mesh, we must bind to the vertex program, set the various input parameters used by the program with the parameter handles, and then enable the particular profile. Underneath the covers of these OpenGL-specific Cg routines, the necessary OpenGL commands are invoked to configure the vertex program with its intended parameter values.

Rather than specifying the parameter value explicitly as with the **`cgGLSetParameter`** routines, the **`cgGLSetStateMatrixParameter`** call binds the current composition of the modelview and projection matrices (specified earlier by **`gluLookAt`** and **`gluPerspective`** commands respectively) to the **`modelViewProj`** parameter.

One of the really nice things about the CgGL runtime is it saves you from having to know the details of what OpenGL routines are called to configure use of your Cg vertex and fragment programs. Indeed, the required OpenGL commands can very considerably between different profiles.

### 3.4.2.2. Binding, Configuring, and Enabling the Fragment Program

```
  cgGLBindProgram(myCgFragmentProgram);
  checkForCgError("binding fragment program");

  cgGLSetParameter4fv(myCgFragmentParam_ambient, myAmbient);
  checkForCgError("setting ambient");
  cgGLSetParameter4fv(myCgFragmentParam_LMd, myLMd);
  checkForCgError("setting diffuse material");
  cgGLSetParameter4fv(myCgFragmentParam_LMs, myLMs);
  checkForCgError("setting specular material");

  cgGLEnableTextureParameter(myCgFragmentParam_normalMap);
  checkForCgError("enable texture normal map");
  cgGLEnableTextureParameter(myCgFragmentParam_normalizeCube);
  checkForCgError("enable 1st normalize vector cube map");
  cgGLEnableTextureParameter(myCgFragmentParam_normalizeCube2);
  checkForCgError("enable 2nd normalize vector cube map");

  cgGLEnableProfile(myCgFragmentProfile);
  checkForCgError("enabling fragment profile");
```

The fragment program is bound, configured, and enabled in much the same manner with the additional task of enabling texture parameters with **`cgGLEnableTextureParameter`** to ensure the indicated texture objects are bound to the proper texture units.

Without you having to know the details, **`cgGLEnableTextureParameter`** calls **`glActiveTexture`** and **`glBindTexture`** to bind the correct texture object (specified earlier with **`cgGLSetTextureParameter`**) into the compiled fragment program's appropriate texture unit in the manner required for the given profile.

### 3.4.2.3. Render the 2D Mesh

```
  drawFlatPatch(sides, rings);
```

With the vertex and fragment program each configured properly, now render the flat 2D mesh that will be formed into a torus and illuminated with specular and diffuse bump mapping.

### 3.4.2.4. Disable the Profiles and Swap

```
  cgGLDisableProfile(myCgVertexProfile);
  checkForCgError("disabling vertex profile");

  cgGLDisableProfile(myCgFragmentProfile);
  checkForCgError("disabling fragment profile");

  glutSwapBuffers();
}
```

While not strictly necessary for this demo because just one object is rendered per frame, after rendering the 2D mesh, the profiles associated with the vertex program and fragment program are each disabled. This way you could perform conventional OpenGL rendering. After using the OpenGL-specific Cg runtime, be careful not to assume how OpenGL state such as what texture objects are bound to what texture units.

## 3.5. *Keyboard Processing*

Along with the **display** callback, we also forward declared and registered the **keyboard** callback. Now it’s time to see how the demo responses to simple keyboard input.

### 3.5.1. Animating the Eye Position

```
static void advanceAnimation(void)
{
  myEyeAngle += 0.05f;
  if (myEyeAngle > 2*3.14159)
    myEyeAngle -= 2*3.14159;
  glutPostRedisplay();
}
```

In order to animate the changing eye position so the view varies, the **advanceAnimation** callback is registered as the GLUT idle function. The routine advances **myEyeAngle** and posts a request for GLUT to redraw the window with **glutPostRedisplay**. GLUT calls the idle function repeatedly when there are no other events to process.

### 3.5.2. The Keyboard Callback

```
static void keyboard(unsigned char c, int x, int y)
{
  static int animating = 0;

  switch (c) {
  case ' ':
    animating = !animating; /* Toggle */
    glutIdleFunc(animating ? advanceAnimation : NULL);
    break;
```

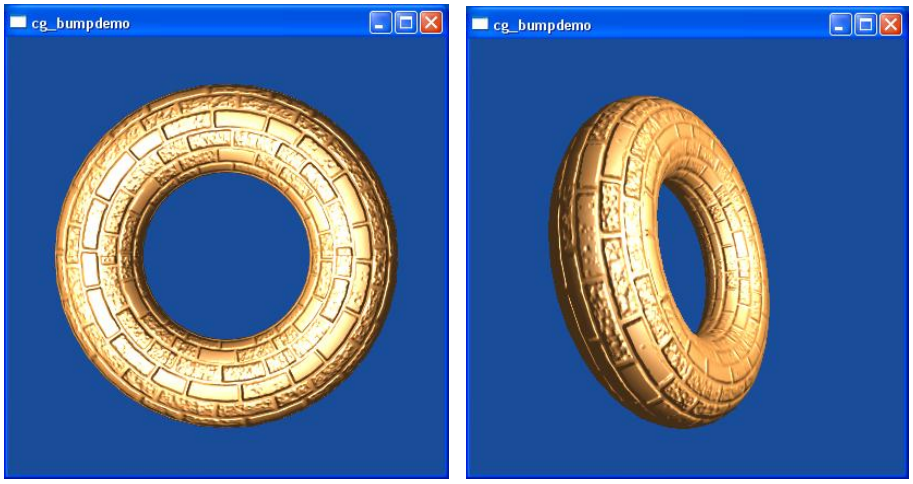
The space bar toggles animation of the scene by registering and de-registering the **advanceAnimation** routine as the idle function.

```
  case 27:  /* Esc key */
    cgDestroyProgram(myCgVertexProgram);
    cgDestroyProgram(myCgFragmentProgram);
    cgDestroyContext(myCgContext);
    exit(0);
    break;
  }
}
```

The Esc key exits the demo. While it is not necessary to do so since the demo is exiting, the calls to **cgDestroyProgram** and **cgDestroyContext** deallocate the Cg runtime objects, along with their associated OpenGL state.

## 4. The Demo in Action

The images below show the rendered bump-mapped torus initially (left) and while animating (right).



## 5. Conclusions

This tutorial presents a complete Cg bump mapping demo written in ANSI C and rendering with OpenGL, relying on two of the actual Cg vertex and fragment programs detailed in *The Cg Tutorial*. I hope this tutorial "fills in the gaps" for those intrepid *Cg Tutorial* readers now inspired to integrate Cg technology into their graphics application. The **`cg_bumpdemo`** demo works on ATI and NVIDIA GPUs (and GPUs from any other vendor that support the standard, multi-vendor vertex and fragment program extensions). The demo is cross-platform as well, supporting Windows, OS X, and Linux systems.

The time you invest integrating the Cg runtime to your graphics application is time well spent because of the productivity and cross-platform support you unleash by writing shaders in Cg rather than resorting to low-level 3D rendering commands or a high-level shading language tied to a particular 3D API. With Cg, you can write shaders that work with two implementations of the same basic language (Cg & HLSL), two 3D rendering programming interfaces (OpenGL & Direct3D), three operating systems (Windows, OS X, and Linux), and the two major GPU vendors (ATI & NVIDIA—and any other vendors supporting DirectX 9-level graphics functionality).

Finally, Cg has evolved considerably since Randy and I wrote *The Cg Tutorial*. Cg 1.2 introduced a "sub-shader" facility allowing you to write shaders in Cg in a more modular fashion. And be sure to explore Cg 1.4's updated implementation of the CgFX meta-shader format (compatible with Microsoft's DirectX 9 FX format) to encapsulate non-programmable state, semantics, hardware-dependent rendering techniques, and support for multiple passes.

## Exercises

Just as *The Cg Tutorial* provides exercises at the end of each chapter, here are some exercises to help you expand on what you've learned.

### *Improving the Shading*

1. Support two lights. You'll need a second light position uniform parameter and your updated vertex program must output a second tangent-space light position. Example 5-4 in *The Cg Tutorial* will give you some ideas for supporting multiple lights. However, Example 5-4 is for two *per-vertex* lights; for this exercise, you want two *per-fragment* lights combined with bump mapping. *Hint:* If you add multiple lights, you might want to adjust down the values of `ambient`, `LMd`, and `LMs` to avoid an "over bright" scene.
2. Support a *positional* light (the current light is directional). Add controls so you can interactively position the light in the "hole" of the torus. Section 5.5 of *The Cg Tutorial* briefly explains the distinction between directional and positional lights.
3. Add geometric self-shadowing to the fragment program.
    a. Clamp the specular to zero if the *z* component of the tangent-space light direction is non-positive to better simulate self-shadowing (this is a situation where the light is "below" the horizon of the torus surface). See section 8.5.3 of *The Cg Tutorial* for details about geometric self-shadowing.
    b. Further tweak the geometric self-shadowing. Instead of clamping, modulate with `saturate(8*lightDirection.z)` so specular highlights don't "wink off" when self-shadowing occurs but rather drop off. When the scene animates, which approach looks better?
4. Change the specular exponent computation to use the `pow` standard library function instead of successive multiplication (you'll find `pow` is only available on more recent DirectX 9-class profiles such as `arbfp1` and `fp30`, not `fp20`). Provide the specular exponent as a uniform parameter to the fragment program.
5. Instead of using normalization cube maps, use the `normalize` standard library routine? Does the lighting change much? Does the performance change?
6. Rather than compute the tangent-space half-angle vector at each vertex and interpolate the half-angle for each fragment, compute the view vector at each vertex; then compute the half-angle at each fragment (by normalizing the sum of the interpolated normalized light vector and the interpolated normalized view vector). Does the lighting change much? Does the performance change?
7. **Advanced:** Read section 8.4 of *The Cg Tutorial* and implement bump mapping on an arbitrary textured polygonal mesh. Implement this approach to bump map an arbitrary textured model.

8. **Advanced:** Read section 9.4 of *The Cg Tutorial* and combine bump mapping with shadow mapping.

## Improving the Cg Runtime Usage

9. Provide command line options to specify what file names contain the vertex and fragment programs.
10. Provide better diagnostic messages when errors occur.
11. Use the Cg runtime to query the uniform parameter names and then prompt the user for values for the various parameters (rather than having the parameter names and values hard coded in the program itself).
12. Rather than using global variables for each vertex and fragment program object, support loading a set of vertex and fragment programs and allow the user to select the current vertex and current fragment program from an interactive menu.

## Appendix A: C8E6v_torus.cg Vertex Program

```
void C8E6v_torus(float2 parametric : POSITION,

             out float4 position       : POSITION,
             out float2 oTexCoord      : TEXCOORD0,
             out float3 lightDirection : TEXCOORD1,
             out float3 halfAngle      : TEXCOORD2,

         uniform float3 lightPosition,  // Object-space
         uniform float3 eyePosition,    // Object-space
         uniform float4x4 modelViewProj,
         uniform float2 torusInfo)
{
  const float pi2 = 6.28318530;  // 2 times Pi
  // Stetch texture coordinates counterclockwise
  // over torus to repeat normal map in 6 by 2 pattern
  float M = torusInfo[0];
  float N = torusInfo[1];
  oTexCoord = parametric * float2(-6, 2);
  // Compute torus position from its parameteric equation
  float cosS, sinS;
  sincos(pi2 * parametric.x, sinS, cosS);
  float cosT, sinT;
  sincos(pi2 * parametric.y, sinT, cosT);
  float3 torusPosition = float3((M + N * cosT) * cosS,
                                (M + N * cosT) * sinS,
                                N * sinT);
  position = mul(modelViewProj, float4(torusPosition, 1));
  // Compute per-vertex rotation matrix
  float3 dPds = float3(-sinS*(M+N*cosT), cosS*(M+N*cosT), 0);
  float3 norm_dPds = normalize(dPds);
  float3 normal = float3(cosS * cosT, sinS * cosT, sinT);
  float3 dPdt = cross(normal, norm_dPds);
  float3x3 rotation = float3x3(norm_dPds,
                               dPdt,
                               normal);
  // Rotate object-space vectors to texture space
  float3 eyeDirection = eyePosition - torusPosition;
  lightDirection = lightPosition - torusPosition;
  lightDirection = mul(rotation, lightDirection);
  eyeDirection = mul(rotation, eyeDirection);
  halfAngle = normalize(normalize(lightDirection) +
                        normalize(eyeDirection));
}
```

## Appendix B: C8E4f_specSurf.cg Fragment Program

```
float3 expand(float3 v) { return (v-0.5)*2; }

void C8E4f_specSurf(float2 normalMapTexCoord : TEXCOORD0,
                    float3 lightDirection    : TEXCOORD1,
                    float3 halfAngle         : TEXCOORD2,

               out float4 color : COLOR,

           uniform float  ambient,
           uniform float4 LMd, // Light-material diffuse
           uniform float4 LMs, // Light-material specular
           uniform sampler2D   normalMap,
           uniform samplerCUBE normalizeCube,
           uniform samplerCUBE normalizeCube2)
{
  // Fetch and expand range-compressed normal
  float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
  float3 normal = expand(normalTex);
  // Fetch and expand normalized light vector
  float3 normLightDirTex = texCUBE(normalizeCube,
                                   lightDirection).xyz;
  float3 normLightDir = expand(normLightDirTex);
  // Fetch and expand normalized half-angle vector
  float3 normHalfAngleTex = texCUBE(normalizeCube2,
                                    halfAngle).xyz;
  float3 normHalfAngle = expand(normHalfAngleTex);

  // Compute diffuse and specular lighting dot products
  float diffuse = saturate(dot(normal, normLightDir));
  float specular = saturate(dot(normal, normHalfAngle));
  // Successive multiplies to raise specular to 8th power
  float specular2 = specular*specular;
  float specular4 = specular2*specular2;
  float specular8 = specular4*specular4;

  color = LMd*(ambient+diffuse) + LMs*specular8;
}
```