CG Programming Tutorial

CIS 665: GPU Programming and Architecture
Instructor: Gary Katz
TA: Joseph Kider

CG Tutorial
- [http://www.seas.upenn.edu/~kiderj/CIS665/](http://www.seas.upenn.edu/~kiderj/CIS665/) (linked on blackboard)
- Slides, links, more details of what I am talking about today.

CG Tutorial (thanks too…)
- Slide information sources:
  - Suresh Venkatasubramanian
    - (RenderTexture Tutorial)
  - Paul Kanyuk
    - Cg ShadingTutorial (Open GL)
  - Mark Harris (Nvidia)
    - SIGGRAPH 2005 (Mapping Computational Concepts to the GPU)
  - Nvidia Corporation
    - Teaching CG
  - Dominik Goddeke’s tutorial

Overview
- **Introduction**
  a. What is CG
  b. Hardware requirements
  c. Software requirements
- **Setting up OpenGL**
  a. GLUT
  b. OpenGL extensions
- **Creating a simple shader with the Cg shading language**
  a. Setting up the Cg runtime
  b. Change color of a box with fragment shader (Demo)
  c. Overview of data: float3, float4, COLOR, wp
- **Arrays = textures**
  a. Creating arrays on the CPU
  b. Creating floating point textures on the GPU
  c. One-to-one mapping from array indices to texture coordinates
  d. Using textures as render targets (FBOs)
  e. Demo Program
- **GPU Transferring Data**
  a. Transferring data from CPU arrays to GPU textures
  b. Transferring data from GPU textures to CPU arrays
  c. Preparing the computational kernel
  d. Setting input arrays / textures
  e. Setting output arrays / textures
  f. Performing the computation
- **GPUGPU concept 4: Feedback**
  a. Multiple rendering passes
  b. The ping pong technique
Introduction: What is CG?

- Cg is an open-source high-level shading language to make graphics programming faster and easier.
- Cg replaces assembly code with a C-like language and a compiler.
- Cg was developed in close collaboration with Microsoft and is syntactically equivalent to HLSL, the shading language in DirectX 9.
- Cg is cross-API (OpenGL & DirectX) and cross-platform (Windows, Linux, and Mac OS).

Introduction: How CG works?

- Shaders are created.
- These shaders are used for modeling in Digital Content Creation (DCC) applications or rendering in other applications.
- The Cg compiler compiles the shaders to a variety of target platforms, including APIs, OSes, and GPUs.
- Spoiler Alert! Porting CG is a pain sometimes since many features are hardware dependent.

Introduction: What does CG look like?

```
Cg

float3 diffuse = powf(powf(0.5, dot(PI, V)), phongExp).x * powf(powf(0.5, dot(PI, V)), phongExp).y * powf(powf(0.5, dot(PI, V)), phongExp).z;

Shading Language (RenderMan™)

color outColor = phong(NV, phongExp);
C1 = Cl + (fSpecularColor * 
(ambientLight + DiffuseLight)) * SpecularColor
    + cSpecular;
```

Introduction: Hardware Requirements

- You will need at least a NVIDIA GeForce 6800 or an ATI RADEON xwhatever graphics card... preferably Nvidia...
- Older GPUs do not provide the features (most importantly, single precision floating point data storage and computation) which we require.
- The CUDA language can only be run on the 8800 cards and the corresponding Quadro cards. The emulator runs on the CPU and does not require a specific card. I am not expecting anyone to complete the homework on the 8800 cards. I am expecting the 8800 card we have will be used by teams that are completing their final project in CUDA.
Introduction Software Requirements

- Again links all on my site... and basic directions what goes where...
- Visual Studio 2005 (preferable)
  - (you can use cygwin, eclipse, g++)
- CG Toolkit 1.5
- GLUT
- GLEW
- Up to date Graphics Drivers!!!
  - Nvidia: 91.*, ATI: Catalyst Software Suite

Introduction: Lab

- No Graphics card? No Money?
- Don’t fret Moore Lab 100B and (HMS is coming) is set up with the proper software and Nvidia 6800s for the Homework assignments

Overview

1. Introduction
   a. What is CG
   b. Hardware requirements
   c. Software requirements
2. Setting up OpenGL
   a. GLUT
   b. OpenGL extensions
3. Creating a simple shader with the Cg shading language
   a. Setting up the Cg runtime
   b. Change color of a box with fragment shader (Demo)
   c. Overview of data: float3, float4, COLOR, wpos
4. Arrays = textures
   a. Creating arrays on the CPU
   b. Creating floating point textures on the GPU
   c. One-to-one mapping from array indices to texture coordinates
   d. Using textures as render targets (FBOs)
   e. Demo Program
5. GPGPU Transferring Data:
   a. Transferring data from GPU arrays to GPU textures
   b. Transferring data from GPU textures to GPU arrays
   c. Preparing the computational kernel
   d. Setting input arrays / textures
   e. Setting output arrays / textures
   f. Performing the computation
6. GPGPU concept 4: Feedback
   a. Multiple rendering passes
   b. The ping pong technique

Setting up OpenGL: GLUT

- GLUT, the OpenGL Utility Toolkit, provides functions to handle window events, create simple menus etc
- Here, we just use it to set up a valid OpenGL context (allowing us access to the graphics hardware through the GL API later on) with as few code lines as possible. Additionally, this approach is completely independent of the window system that is actually running on the computer

```c
// include the GLUT header file
#include <GL/glut.h>

// call this and pass the command line arguments from main()
void initGL(int argc, char **argv) {
    glutInit(&argc, argv);
    glutCreateWindow("GLUT TEST");
}
```
Setting up OpenGL: GLEW

- The small tool `glewinfo` that ships with GLEW, or any other OpenGL extension viewer, or even OpenGL itself can be used to check if the hardware and driver support a given extension.
- Obtaining pointers to the functions the extensions define is an advanced issue, so in this example, we use GLEW as an extension loading library that wraps everything we need up nicely with a minimalistic interface.

```c
#include <CG/cgGL.h>

// CG global parameters
CGcontext cgContext;

// CG program data
CGPROFILE cgProfile;

CGPARAMETER CGparam;
CGPARAMETER CGparam2;
CGPARAMETER CGparam3;
CGPARAMETER CGparam4;

// Cg context is the entry point for the Cg runtime, since we want to program the
// fragment pipeline, we need a fragment profile (Cg is profile-based) and a
// program container for the program we just wrote.
```

Simple Shader: Setting up CG

This subsection describes how to set up the Cg runtime in an OpenGL application. First, we need to include the Cg headers (it is sufficient to include `<Cg/cgGL.h>`) and add the Cg libraries to our compiler and linker options. Then, we declare some variables:

```c
// CG global parameters
cgContext = cgCreateContext(); // Create cgContext.
theProfile = cgCreateProfile(CG_GL_FRAGMENT);
assert(NULL != theProfile); // Must get a valid profile!

// CG runtime options
cgSetRuntimeOption(CG_PROFILE, theProfile);

// Create the program
theProgram = cgCreateProgramFromSource(cgContext, "main.cg", theProfile, NULL, NULL);
```

Setting up CG: Parameters

```c
if (theProgram != NULL) {
    cgMainProgram = theProgram;
    input1 = cgInputParameter(cgMainProgram, "input1");
    assert(NULL != input1);
    input1 = NULL;
}
```

```c
if (input1 == NULL) {
    fprintf(stderr, "Could not load Cg program! FATAL ERROR!
    exit(1);
```
Setting up Cg: Vertex Processor

- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex
  - Cannot read info from other vertices
  - Can only read a small constant memory
- Latest GPUs: Vertex Texture Fetch
  - Random access memory for vertices
  - Gather (But not from the vertex stream itself)

Setting up Cg: Fragment Processor

- Fully programmable (SIMD)
- Processes 4-component vectors (RGBA / XYZW)
- Random access memory read (textures)
- Capable of gather but not scatter
  - RAM read (texture fetch), but no RAM write
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
  - More fragment pipelines than vertex pipelines
  - Direct output (fragment processor is at end of pipeline)

Setting up Cg: Demos

- Green Sphere
- 2 color Box Demo
- Normal Vertex Sphere
- Plastic” Per-Vertex Shading

Setting up Cg: Data Structures

- float4, float3 (packed arrays /not vectors)
- in : variables coming in from pipeline
- out: variables going out to pipeline
- WPOS, position: positional vectors
- Uniform int, float: input values
- in float2 coords: TEXCOORD0 : texture coords
- tex2d, sampler2d, samplerRECT : input textures

WARNING:
Make sure you are consistent with recs and 2ds when setting up textures!!!
Overview

1 Introduction
   a. What is CG
   b. Hardware requirements
   c. Software requirements
2 Setting up OpenGL
   a. GLUT
   b. OpenGL extensions
3 Creating a simple shader with the Cg shading language
   a. Setting up the runtime
   b. Change color of a box with fragment shader (Demo)
   c. Overview of data: float3, float4, COLOR, wpos
4 Arrays = textures
   a. Creating arrays on the CPU
   b. Creating floating point textures on the GPU
   c. One-to-one mapping from array index to texture coordinates
   d. Using textures as render targets (FBOs)
   e. Demo Program
5 GPGPU Transferring Data:
   a. Transferring data from CPU arrays to GPU textures
   b. Transferring data from GPU textures to CPU arrays
   c. Preparing the computational kernel
   d. Setting input arrays / textures
   e. Setting output arrays / textures
   f. Performing the computation
6 GPGPU concept 4: Feedback
   a. Multiple rendering passes
   b. The ping pong technique

Textures: C++ Arrays (CPU)

○ Creating arrays on the CPU
○ One option to hold data for GPGPU calculations

```c
float dataY = (float)malloc(sizeOf(float));
float dataX = (float)malloc(sizeOf(float));
float alpha;
```

Another option for rendering is to draw geometry and use that as the input data to the textures used more for advanced rendering effects.

Textures: OpenGL

○ This gets complicated fast...
○ Look at glTexImage2D
    - Texture_target (next slide)
    - Internal format (next slide)
    - texSize, texSize (width and height of the texture)
    - 0: turns off borders for our texture
    - Texture_format: chooses the number of channels
      - GL_Float: Float texture (nothing to do with the precision of the values)
      - 0 or NULL: We do not want to specify texture data right now...

// create a new texture name
GLuint textures[2];
GLuint textureX = glGenTextures(1, &textures[0]);
GLuint textureY = glGenTextures(1, &textures[1]);
// bind the texture name to a texture target
glTexImage2D(TEXTURE_TARGET, 0, TEXTURE_INTERNAL_FORMAT, texSize, texSize, 0, TEXTURE_FORMAT, TEXTURE_TYPE, NULL);
// (optional for float textures)
glTexParameteri(TEXTURE_TARGET, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexParameteri(TEXTURE_TARGET, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// call texture copy function
glCopyTexImage2D(TEXTURE_TARGET, 0, TEXTURE_INTERNAL_FORMAT, textureX, 0, 0);
// this will set the data in the new textures
glTexImage2D(TEXTURE_TARGET, 0, TEXTURE_INTERNAL_FORMAT, texSize, texSize, 0, TEXTURE_FORMAT, TEXTURE_TYPE, NULL);

Textures: Formats

○ On the GPU, we use floating point textures to store the data
○ A variety of different so-called texture targets available

### Texture Target | Texture Format
--- | ---
GL_TEXTURE_2D | GL_FLOAT
GL_TEXTURE_3D | GL_FLOAT
GL_TEXTURE_CUBE_MAP | GL_FLOAT

---

### Dimensions
- Dimensions are specified in powers of two. Sample size is 4 times the power of two. OpenGL requires 4 times a power of two.
- GL_TEXTURE_2D
- GL_TEXTURE_3D
- GL_TEXTURE_CUBE_MAP

### Internal texture format
- GPUs allow for the simultaneous processing of scalars, tupels, tripels or four-tupels of data
- Precision of data: GL_FLOAT, GL_R32_NV, GL_R16, GL_RGB, GL_RGB16, GL_RGBA ...

- More explanation on website tutorial
- ATI warning – here is where you need to specify ATI extensions
Mapping textures

- Later we update our data stored in textures by a rendering operation.
- To be able to control exactly which data elements we compute or access from texture memory, we will need to choose a special projection that maps from the 3D world (world or model coordinate space) to the 2D screen (screen or display coordinate space), and additionally a 1:1 mapping between pixels (which we want to render to) and texels (which we access data from).
- The key to success here is to choose an orthogonal projection and a proper viewport that will enable a one to one mapping between geometry coordinates.
- (add this to your reshape, init, and initFBO methods)

```c
// reshape up to 2D projection of screen-geometry mapping
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, width, 0, height, -1, 1);
```

Frame Buffer Objects: (FBO)

To use this extension and to turn off the traditional framebuffer and use an offscreen buffer (surface) for our calculations, a few lines of code suffice. Note that binding FBO number 0 will restore the window-system specific framebuffer at any time.

```c
void initFBO(void) {
    // create FBO (off-screen framebuffer)
    glGenFramebuffers(1, &fbo);
    glBindFramebuffer(GL_FRAMEBUFFER, fbo);
    // create 2D texture
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_2D, tex);
    // set texture for attachment
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
    // set texture as color attachment
    glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, tex, 0);
    // set render target
    glBindFramebuffer(GL_FRAMEBUFFER, fbo);
}
```

- The framebuffer object extension provides a very narrow interface to render to a texture. To use a texture as render target, we have to attach the texture to the FBO.
- drawback is: Textures are either read-only or write-only (important later)

Using Textures as Render Targets

- the traditional end point of every rendering operation is the frame buffer, a special chunk of graphics memory from which the image that appears on the display is read.
- Problem! : the data will always be clamped to the range of [0/255; 255/255] once it reaches the framebuffer. What to do?
- cumbersome arithmetic that maps the sign-mantissa-exponent data format of an IEEE 32-bit floating point value into the four 8-bit channels ???
- OpenGL extension called EXT_framebuffer_object allows us to use an offscreen buffer as the target for rendering operations such as our vector calculations, providing full precision and removing all the unwanted clamping issues. The commonly used abbreviation is FBO, short for framebuffer object.

Using FBOs: DEMO

- HelloGPGPU Demo
Overview

1 Introduction
   a. What is CG
   b. Hardware requirements
   c. Software requirements

2 Setting up OpenGL
   a. GLUT
   b. OpenGL extensions

3 Creating a simple shader with the Cg shading language
   a. Setting up the CG runtime
   b. Setting up the fragment shader (Demo)
   c. Overview of data: float3, float4, COLOR, upos

4 Arrays = textures
   a. Creating arrays in the CPU
   b. Creating arrays on the GPU
   c. One-to-one mapping from array index to texture coordinates
   d. Using textures as render targets (FBOs)

5 GPGPU Transferring Data:
   a. Transferring data from GPU textures to CPU arrays
   b. Transferring data from GPU textures / textures to CPU arrays
   c. Setting up the computational kernel
   d. Setting up arrays / textures
   e. Setting up output arrays / textures
   f. Performing the computation

6 GPGPU concept 4: Feedback
   a. Multiple rendering passes
   b. The ping pong technique

Transferring data from GPU textures to CPU arrays

- Many times you want the actual values that you calculated back, there are 2 ways to do this

  ```
  glBindTexture(GL_TEXTURE_1D, tex1);
  glBindTexture(GL_TEXTURE_2D, tex2);
  void* dataX = (void*) malloc(sizeof(float) * length);
  void* dataY = (void*) malloc(sizeof(float) * length);
  ```

  In order to transfer data (like the two vectors dataX and dataY) we created previously to a texture, we have to bind the texture to a texture target and schedule the data for transfer with an OpenGL (note: NVIDIA Code)

  ```
  glEnable(GL_TEXTURE_1D, true);
  glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, width, 0, GL_RGBA, GL_FLOAT, dataX);
  glEnable(GL_TEXTURE_2D, true);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_FLOAT, dataY);
  ```

- Again not only method, if you rather do rendering rather than GPGPU computations draw geometry to the buffer directly as follows:

  ```
  // Draw to buffer
  glDrawArrays(GL_TRIANGLES, 0, 3);
  // Draw to texture
  glDrawArrays(GL_TRIANGLES, 0, 3);
  ```

Transferring data from GPU textures to QUADS

- Other time you really just want to see the mess you created on the screen

  ```
  // Render to a QUAD
  glEnable(GL_TEXTURE_2D, true);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_FLOAT, dataY);
  glEnable(GL_TEXTURE_1D, true);
  glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, width, 0, GL_RGBA, GL_FLOAT, dataX);
  glDrawArrays(GL_TRIANGLES, 0, 3);
  ```

- To do this you have to render a QUAD

  ```
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glEnable(GL_TEXTURE_1D, true);
  glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, width, 0, GL_RGBA, GL_FLOAT, dataX);
  glEnable(GL_TEXTURE_2D, true);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_FLOAT, dataY);
  glDrawArrays(GL_TRIANGLES, 0, 3);
  ```
Preparing the computational kernel
setting up input textures/arrays

```c
// Bind the texture as input to the shader.
oglGenTextures(1, &inputTexture[0]);
oglBindTexture(0, inputTexture[0]);
// In order to execute fragment programs, we need to generate pixels.
// Creating a quad the size of our viewport (not drawn) generates a
// fragment for every pixel of our destination texture. Each fragment
// is processed identically by the fragment program. Notice that in
// the `color()` function, below, we have set the fragment to
// orthographic, and the fragment dimensions to [-1,1]. Thus, our
// viewport-sized quad vertices are at [-1,-1], [1,-1], [1,1], and
// [-1,1] the corners of the viewport.
oglEnable(32024);
oglTexCoord2f(0, 0); glVertex3f(1, 1, 0.5);
oglTexCoord2f(1, 0); glVertex3f(-1, 1, 0.5);
oglTexCoord2f(1, 1); glVertex3f(-1, -1, 0.5);
oglTexCoord2f(0, 1); glVertex3f(1, -1, 0.5);
oglDisable(32024);
oglEnableTextureParameter(1, true);
``` 

Performing a computation

- Let us briefly recall what we did so far.
- We enabled a 1:1 mapping between the target pixels, the texture coordinates and the geometry we are about to draw.
- We also prepared a fragment shader we want to execute for each fragment.
- All that remains to be done is: Render a "suitable geometry" that ensures that our fragment shader is executed for each data element we stored in the target texture.
- In other words, we make sure that each data item is transformed uniquely into a fragment.
- Given our projection and viewport settings, this is embarrassingly easy: All we need is a filled quad.

Setting output arrays / textures

Defining the output array (the left side of the equation) is essentially the same operation like the one we discussed to transfer data to a texture already attached to our FBO. Simple pointer manipulation by means of GL calls is all we need. In other words, we simply redirect the output: If we did not do so yet, we attach the target texture to our FBO and use standard GL calls to use it as the render target:

```c
// attach target texture to first attachment point
glFramebufferTexture2D(GL_FRAMEBUFFER, 
    GL_COLOR_ATTACHMENT0, 
    textureTarget, textureID[0], 0);
// set the texture as render target
glDrawBuffer(GL_COLOR_ATTACHMENT0);  
```
Multiple rendering passes

- In a proper application, the result is typically used as input for a subsequent computation.
- On the GPU, this means we perform another rendering pass and bind different input and output textures, eventually a different kernel etc.
- The most important ingredient for this kind of multipass rendering is the ping pong technique.

The ping pong technique

- **Ping pong** is a technique to alternately use the output of a given rendering pass as input in the next one.
- Let's look at this operation: \( y_{\text{new}} = y_{\text{old}} + \alpha \cdot x \)
- this means that we swap the role of the two textures \( y_{\text{new}} \) and \( y_{\text{old}} \), since we do not need the values in \( y_{\text{old}} \) any more once the new values have been computed.
- There are three possible ways to implement this kind of data reuse (take a look at Simon Green’s FBO slides for additional material on this, link posted on the url):

```cpp
// Fbo shaders [multitex referencing y_old and y_new
uniform float alpha;

// ping pong alignment setup
for (i = 0; i < 4; ++i) {
    glActiveTexture(GL_TEXTURE0 + i);
    glBindTexture(GL_TEXTURE_2D, texture[i]);
    glVertexAttribPointer(i, 1, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(i);
    glActiveTexture(GL_TEXTURE0 + (i + 3) % 4);
    glBindTexture(GL_TEXTURE_2D, texture[i + 3 % 4]);
    glVertexAttribPointer(i, 1, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(i);
}

// enable fragment profile, set program [...] // enable texture (read-only) and write parameter [...] // iterate computation several times

// set render destination
glActiveTexture(GL_TEXTURE0 + 4);
// enable texture y_gls (read-only)
glActiveTexture(GL_TEXTURE0 + 5);
// enable texture y_gls (read-only)
// enable multitexture cullzot of plat quad
// enable multitexture draw only source becomes // write-only target and the other two read only!
```
Closing thoughts…

- Best to just hack away
- I have some simple debugging code imbedded in the demos ... best to take a look at it and use it ... debugging on the GPU is not explicit
- Problems 1 and 2: Best to start from `runtime_ogl_vertex_(fragment/vertex) examples`
- Problem 3: Best to start from Demo2 HelloGPGPU example
- Next Homework GPGPU stuff: Best to start from DEMO3