CG Programming Tutorial

CIS 665

GPU Programming and Architecture

Joseph Kider

CG Tutorial

- http://www.seas.upenn.edu/~cis665/
  - Schedule and resource pages
- 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

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 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

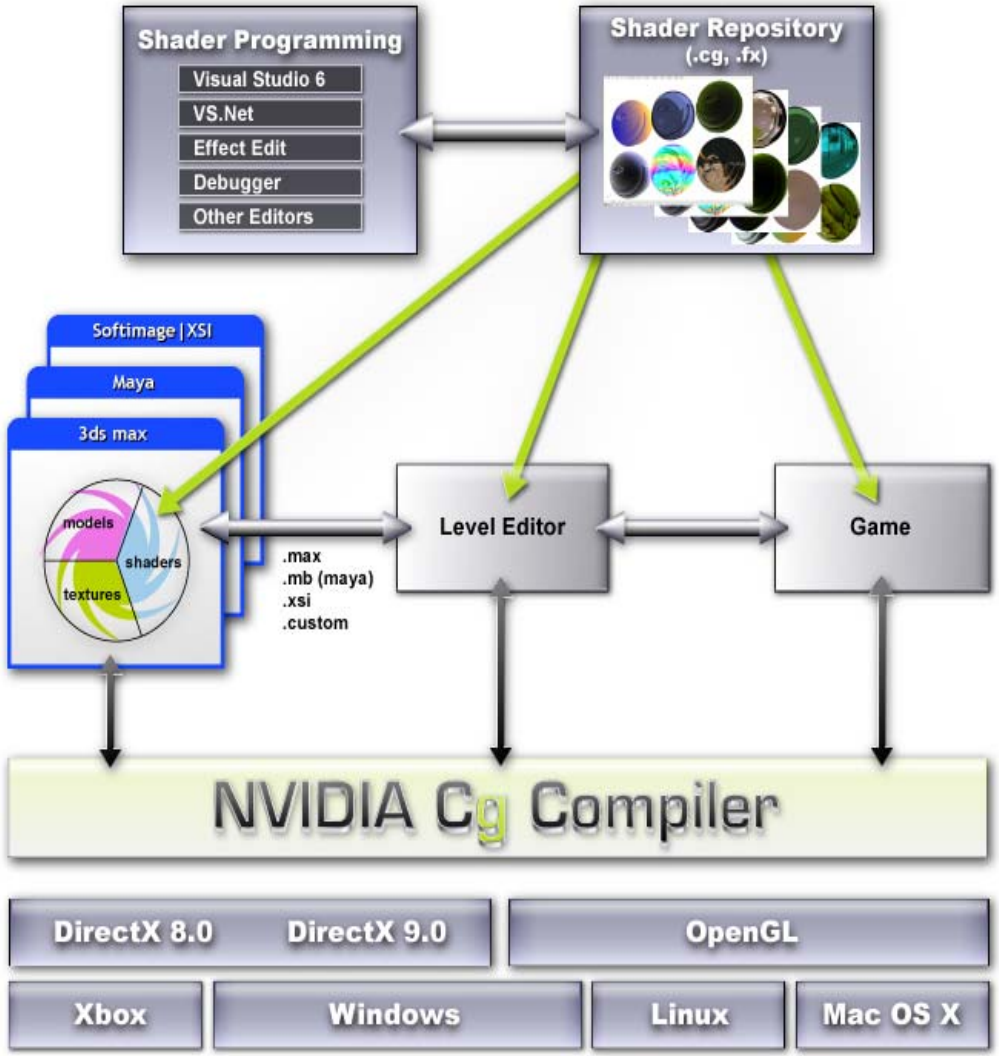
6 GPGPU concept 4: Feedback
- a. Multiple rendering passes
- b. The ping pong technique

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 dependant.



Introduction: What does CG look like?

Assembly

```
...
RSQR  R0.x, R0.x;
MULR  R0.xyz, R0.xxxx, R4.xyzz;
MOVR  R5.xyz, -R0.xyzz;
MOVR  R3.xyz, -R3.xyzz;
DP3R  R3.x, R0.xyzz, R3.xyzz;
SLTR  R4.x, R3.x, {0.000000}.x;
ADDR  R3.x, {1.000000}.x, -R4.x;
MULR  R3.xyz, R3.xxxx, R5.xyzz;
MULR  R0.xyz, R0.xyzz, R4.xxxx;
ADDR  R0.xyz, R0.xyzz, R3.xyzz;
DP3R  R1.x, R0.xyzz, R1.xyzz;
MAXR  R1.x, {0.000000}.x, R1.x;
LG2R  R1.x, R1.x;
MULR  R1.x, {10.000000}.x, R1.x;
EX2R  R1.x, R1.x;
MOVR  R1.xyz, R1.xxxx;
MULR  R1.xyz, {0.900000, 0.800000,
      1.000000}.xyzz, R1.xyzz;
DP3R  R0.x, R0.xyzz, R2.xyzz;
MAXR  R0.x, {0.000000}.x, R0.x;
MOVR  R0.xyz, R0.xxxx;
ADDR  R0.xyz, {0.100000, 0.100000,
      0.100000}.xyzz, R0.xyzz;
MULR  R0.xyz, {1.000000, 0.800000,
      0.800000}.xyzz, R0.xyzz;
ADDR  R1.xyz, R0.xyzz, R1.xyzz;
```

Cg

```
…
float3 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;
float3 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;
…
```

Shading Language (RenderMan™)

```
…
color cSpec = phong(Nf,V,phongExp);
Ci = Oi * (FinalColor = DiffuseColor *
    (AmbientLight + DiffuseLight)) + SpecularColor
    * cSpec;
…
```

Introduction: Hardware Requirements

- You will need at least a NVIDIA GeForce 5800 or an ATI RADEON x9800 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 for following homeworks and the final project.

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++)
- At least CG Toolkit 1.5 (2+ is preferable)
- GLUT
- GLEW
- Up to date Graphics Drivers!!!
  - Go to the Nvdia Driver page and ATI: Catalyst Software Suite

Introduction: Lab

- No Graphics card? No Money?
- Don’t fret Moore Lab 100B and (HMS lab 8800s) is set up with the proper software and Nvidia 6800s for the Homework assignments …. I hope!

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 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

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

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

// call this and pass the command line arguments from main()
void initGLUT(int argc, char **argv) {
    glutInit ( &argc, argv );
    glutCreateWindow(" GLUT TESTS");
}
```

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:

```
void initGLEW (void) {
    // init GLEW, obtain function pointers
    int err = glewInit();
    // Warning: This does not check if all extensions used
    // in a given implementation are actually supported.
    // Function entry points created by glewInit() will be
    // NULL in that case!
    if (GLEW_OK != err) {
        printf((char*)glewGetErrorString(err));
        exit(ERROR_GLEW);
    }
}
```

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 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

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:

```
//CG Global Parameters
CGprogram   demo;
CGcontext   cgContext;
CGprofile   theProfile;
CGparameter input1;
```

The **CGcontext** 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. For the sake of simplicity, we also declare three handles to the parameters we use in the shader that are not bound to any semantics, and we use a global variable that contains the shader source we just wrote.

Setting up CG: Parameters

```
void initCGParams()
{
    cgContext = cgCreateContext(); // Create cgContext.

    theProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    assert(theProfile != CG_PROFILE_UNKNOWN);
    cgGLSetOptimalOptions(theProfile);
    cgSetErrorCallback(cgErrorCallback);

    demo = cgCreateProgramFromFile(cgContext, CG_SOURCE,
                                   "demo.cg", theProfile, NULL,NULL);

    if(demo != NULL)  {
        cgGLLoadProgram(demo);
        input1 = cgGetNamedParameter(demo, "mChangeColorPosition");
        assert(input1 != NULL);
    }
    else {
        fprintf(stderr, "Could not load Cg program ! FATAL ERROR!\n");
        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)

```
                                  Entry Point to Cg Program.  Followed by a parameter list.
void main(float4 Pobject      : POSITION,   <= Varying Input Parameter (per-vertex value)
          uniform float4x4 ModelViewProj,   <= Uniform Input Parameter (constant value)

          out float4 HPosition    : POSITION,  <= Output Parameter (passed to fragment proc)
          out float4 oColor       : COLOR0)    <= Output Parameter (passed to fragment proc)
{
   // compute homogeneous position of vertex for rasterizer
   HPosition = mul(ModelViewProj, Pobject);     <= Matrix by Vector multiplication.
                                                   Output Position transformed and set.
   oColor = float4(0,1,0,1);                    <= Output Color Set to GREEN.
}
   Body of the Cg Program, uses input params to compute out params.
```

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)

Entry Point to Cg Program. Followed by a parameter list.

```
void main( float4 iColor      : COLOR0,
           float3 iNormal     : TEXCOORD0,

           out float4 oColor : COLOR)
{
   oColor =  float4(normalize(iNormal),1) * iColor;
}
```

<= Varying Input Parameters (interpolated from the output of the vertex program)

<= Output Parameter passed to GPU Raster Operations

<= The output color is set to the product of the normalized texcoord0 and color0 input.

Body of the Cg Program, uses input params to compute out params.

Setting up CG: Demos

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

Setting up CG: Data Structures

```
float4 main(
        in float4 col : COL0, // CG color information
        in float3 pos : WPOS,   // CG position information
        uniform int mChangeColorPosition
           ) : COLOR
```

- float4, float3 (packed arrays /not vectors)
- in : variables coming in from pipeline
- out: variables going out to pipeline
- WPOS, position: positional vectors
- Uniform int,floats : 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 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 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

```
float* dataY = (float*)malloc(N*sizeof(float));
float* dataX = (float*)malloc(N*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)
  - 0: not to use any mipmap levels for this texture
  - 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 texID;
glGenTextures (1, &texID);
// bind the texture name to a texture target
glBindTexture(texture_target,texID);
// turn off filtering and set proper wrap mode
// (obligatory for float textures atm)
glTexParameteri(texture_target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_T, GL_CLAMP);
// set texenv to replace instead of the default modulate
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
// and allocate graphics memory
glTexImage2D(texture_target, 0, internal_format,
             texSize, texSize, 0, texture_format, GL_FLOAT, 0);
```

Textures: Formats

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

	texture2D	texture rectangle
texture target	GL_TEXTURE_2D	GL_TEXTURE_RECTANGLE_ARB
texture coordinates	Coordinates have to be normalized to the range [0,1] by [0,1], independent of the dimension [0,M] by [0,N] of the texture.	Coordinates are not normalized.
texture dimensions	Dimensions are constrained to powers of two (e.g. 1024 by 512) unless the driver supports the extension ARB_non_power_of_two **or** unless the driver exposes OpenGL 2.0 which alleviates this restriction.	Dimensions can be arbitrary by definition, e.g. 513 by 1025.

- **Internal texture format**. GPUs allow for the simultaneous processing of scalars, tupels, tripels or four-tupels of data
- Precision of data: **GL_FLOAT_R32_NV, GL_R, 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)

```
// viewport for 1:1 pixel=texel=geometry mapping
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, texSize, texSize);
```

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.

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.

```
GLuint fb;

void initFBO(void) {
    // create FBO (off-screen framebuffer)
    glGenFramebuffersEXT(1, &fb);
    // bind offscreen buffer
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
}
```

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          texture_target, texID, 0);
```

- 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 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. 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

Transferring data from CPU arrays to GPU textures

- 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)

```
glBindTexture(texture_target, texID);
glTexSubImage2D(texture_target,0,0,0,texSize,texSize,
                texture_format,GL_FLOAT,data);
```

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

```
glGetIntegerv(GL_DRAW_BUFFER, &_currentDrawbuf); // Save Draw buffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);  // Draw into the first texture

int vp[4];
glGetIntegerv(GL_VIEWPORT, vp);

glViewport(0, 0, texSize, texSize);

 DrawGeometry();

 glDrawBuffer(GL_COLOR_ATTACHMENT1_EXT);//Draw into the second texture
  // now use this as input to a CG program ...
  // use the 2nd buffer as the write buffer
  // remember you can only read or write to a buffer....
```

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(texture_target,texID);
glGetTexImage(texture_target,0,texture_format,GL_FLOAT,data);
```

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0,0,texSize,texSize,texture_format,GL_FLOAT,data);
```

Transferring data from GPU textures to QUADS

- Other time you really just want to see the mess you created on the screen
- To do this you have to render a QUAD

```
 // Now Render it to a QUAD
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0); // disable the frame buffer
glDrawBuffer(_currentDrawbuf);
glBindTexture(GL_TEXTURE_2D, _iTexture[1]);
glEnable(GL_TEXTURE_2D);

// render a full-screen quad textured with the results of our
// computation.  Note that this is not part of the computation: this
// is only the visualization of the results.
glBegin(GL_QUADS);
{
    glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
    glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
    glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
    glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
}
glEnd();

glDisable(GL_TEXTURE_2D);
 glutSwapBuffers();
```

Preparing the computational kernel setting up input textures/arrays

```
glBindTexture(GL_TEXTURE_2D, _iTexture[0]); //read from the first texture

cgGLBindProgram(demo);
cgGLEnableProfile(theProfile);

// bind the scene texture as input to the filter
cgGLSetTextureParameter(_textureParam, _iTexture[0]);
cgGLEnableTextureParameter(_textureParam);

// In order to execute fragment programs, we need to generate pixels.
// Drawing a quad the size of our viewport (see above) generates a
// fragment for every pixel of our destination texture. Each fragment
// is processed identically by the fragment program. Notice that in
// the reshape() function, below, we have set the frustum to
// orthographic, and the frustum 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.
glBegin(GL_QUADS);
{
    glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
    glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
    glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
    glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
}
glEnd();

cgGLDisableTextureParameter(_textureParam);  // Disable the params
cgGLDisableProfile(theProfile);
```

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:

```
// attach target texture to first attachment point
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          texture_target, y_newTexID, 0);
// set the texture as render target
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);
```

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**

```
// make quad filled to hit every pixel/texel
glPolygonMode(GL_FRONT,GL_FILL);
// and render quad
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(texSize, 0.0);
    glVertex2f(texSize, 0.0);
    glTexCoord2f(texSize, texSize);
    glVertex2f(texSize, texSize);
    glTexCoord2f(0.0, texSize);
    glVertex2f(0.0, texSize);
glEnd();
```

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 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

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.
- Lets look at this operation: (**y_new** = **y_old** + *alpha* * **x**)
- this means that we swap the role of the two textures **y_new** and **y_old**, since we do not need the values in **y_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):

```
// two textures identifiers referencing y_old and y_new
GLuint yTexID[2];
// ping pong management vars
int writeTex = 0;
int readTex = 1;
GLenum attachmentpoints[] = { GL_COLOR_ATTACHMENT0_EXT,
                              GL_COLOR_ATTACHMENT1_EXT
                            };
```

The ping pong technique

- During the computation, all we need to do now is to pass the correct value from these two tupels to the corresponding OpenGL calls, and to swap the two index variables after each pass:

```
// attach two textures to FBO
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          attachmentpoints[writeTex],
                          texture_Target, yTexID[writeTex], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          attachmentpoints[readTex],
                          texture_Target, yTexID[readTex], 0);
// enable fragment profile, bind program [...]
// enable texture x (read-only) and uniform parameter [...]
// iterate computation several times
for (int i=0; i<numIterations; i++) {
    // set render destination
    glDrawBuffer (attachmentpoints[writeTex]);
    // enable texture y_old (read-only)
    cgGLSetTextureParameter(yParam, yTexID[readTex]);
    cgGLEnableTextureParameter(yParam);
    // and render multitextured viewport-sized quad
    // swap role of the two textures (read-only source becomes
    // write-only target and the other way round):
    swap();
}
```

The ping pong Demo

- Saxpy Demo

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