Use Cg Language

Adapted from
Suresh Venkatasubramanian
Klaus Mueller
What is Cg?

- Cg stands for “C for Graphics”.
- Developed by Nvidia

- Cg compiles to code that can be executed by a GPU’s programmable fragment or vertex processor

- Cg is “theoretically” cross-platform working for OpenGL and DirectX
What is Cg?

- Can work with OpenGL as well as Direct3D
- CG compiler produces OpenGL or Direct3D code
- OpenGL or Direct3D drivers perform final translation into hardware-executable code
  - core CG runtime library (CG prefix)
  - cgGL and cgD3D libraries
What is Cg used for?
What does Cg look like?

Assembly

```
... 
RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxx, 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.xxx, R5.xyzz;
MULR R0.xyz, R0.xyzz, R4.xxx;
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.xxx;
MULR R1.xyz, {0.900000, 0.800000, 1.000000}.xyz, R1.xyzz;
DP3R R0.x, R0.xyzz, R2.xyzz;
MAXR R0.x, {0.000000}.x, R0.x;
MOVR R0.xyz, R0.xxx;
ADDR R0.xyz, {0.100000, 0.100000, 0.100000}.xyz, R0.xyzz;
MULR R0.xyz, {1.000000, 0.800000, 0.800000}.xyz, 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;
... 
```
Where do I get Cg?


- Download the latest Cg Toolkit:
  - Cg Compiler
  - Cg Runtime
  - Cg User’s Manual
  - Cg Browser
  - Sample Cg Shaders

- The installer setup the appropriate environment variables (PATH, CG_LIB_PATH, CG_INC_PATH).
Compilation

- To interface Cg programs with application:
  - compile the program with appropriate profile
    - dependent on underlying hardware for shader versions
  - range of profiles will grow with GPU advances
    - OpenGL: vp20, vp30, fp30, fp40 …
- Link the program to the application program
- Can perform compilation at
  - compile time (static)
  - runtime (dynamic)
How can I use Cg?

- Your application must call the Cg Runtime to invoke the Cg programs and pass the appropriate parameters.
Cg Runtime

- The Cg Runtime can be more challenging than Cg itself.

- For complete documentation on the Cg Runtime, see the Cg User’s Manual.
Cg Runtime

- Can take advantage of
  - latest profiles
  - optimization of existing profiles
- No dependency issues
  - register names, register allocations

In the following, use OpenGL to illustrate
  - Direct3D similar methods
Preparing a Cg Program

First, create a context:
- context = cgCreateContext()

Compile a program by adding it to the context:
- program = cgCreateProgram(context, programString, profile, name, args)

Loading a program (pass to the 3D API):  
- cgGLLoadProgram(program)
Running a Cg Program

- Executing the profile:
  - cgEnableProfile(CG_PROFILE_ARBVP1)
- Bind the program:
  - cgGLBindProgram(program)
- After binding, the program will execute in subsequent drawing calls
  - for every vertex (for vertex programs)
  - for every fragment (for fragment programs)
  - these programs are often called *shaders*
void CgGLInit()
{
    // Create the fragment program
    fragmentProgram = cgCreateProgramFromFile(
        context, CG_SOURCE, "FragmentProgram.cg",
        fragmentProfile, "FragmentProgram", 0);

    // Load the program
    cgGLLoadProgram(fragmentProgram);

    // Grab some parameters.
    someColor = cgGetNamedParameter(fragmentProgram,
        "SomeColor");
    cgGLSetParameter4fv(someColor, constantColor);
}

// Called to render the scene
void Display()
{
    // Bind the programs
    cgGLBindProgram(fragmentProgram);
    // Draw scene
    // ...
}
Running a Cg Program

- One one vertex / fragment program can be bound at a time
  - the same program will execute unless another program is bound
- Disable a profile by:
  - cgGLDisableProfile()
- Release resources:
  - cgDestroyProgram(program)
  - cgDestroyContext(context)
  - the latter destroys all programs as well
Error Handling

- There are core CG routines that retrieve global error variables:
  - `error = cgGetError()`
  - `cgGetErrorString(error)`
  - `cgSetErrorCallback(MyErrorCallback)`
Anatomy of a Cg Vertex Program

void main(float4 Pobject : POSITION, 
          uniform float4x4 ModelViewProj,
          cut float4 HPosition : POSITION,
          cut float4 oColor : COLOR0)
{
    // compute homogeneous position of vertex for rasterizer
    HPosition = mul(ModelViewProj, Pobject);
    oColor = float4(0,1,0,1);
}

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

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

- Varying Input Parameter (per-vertex value)
- Uniform Input Parameter (constant value)
- Output Parameter (passed to fragment proc)
- Output Parameter (passed to fragment proc)
- Matrix by Vector multiplication.
  Output Position transformed and set.
- Output Color Set to GREEN.
Anatomy of a Cg Fragment Program

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

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

out float4 oColor : COLOR)
{

doColor = 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.
Simple Vertex Program

*Semantics* connect Cg program with graphics pipeline
- here: POSITION and COLOR
Uniform vs. Varying Parameters

- Varying: values vary per vertex or fragment
  - interfaced via semantics
- Uniform: remain constant
  - interfaced via handles

```c
struct C3E1v_Output {
    float4 position  // POSITION;
    float4 color     // COLOR;
};

C3E1v_Output C3E1v_anyColor(float3 position : POSITION,
                             uniform float4 constantColor)
{
    C3E1v_Output OUT;

    OUT.position = float4(position, 0, 1);
    OUT.color = constantColor; // Some RGBA color
    return OUT;
}
```
Passing Parameters

- Assume these shader variables:
  - float4 position : POSITION
  - float4 color : COLOR0

- Get the handle for color by:
  - color = cgGetNamedParameter(program, "IN.color")

- Can set the value for color by:
  - cgGLSetParameter4f(color, 0.5f, 1.0f, 0.5f, 1.0f)

- Uniform variables are set infrequently:
  - example: modelViewMatrix
Passing Parameters

- Set other variables via OpenGL semantics:
  - `glVertex`, `glColor`, `glTexCoord`, `glNormal`, ...

- Example: rendering a triangle with OpenGL:

```c
glBegin(GL_TRIANGLES);
    glVertex( 0.8, 0.8);
    glVertex(-0.8, 0.8);
    glVertex( 0.0, -0.8);
 glEnd();
```

- `glVertex` affects POSITION semantics and updates/sets parameter in vertex shader
Example 1

- Vertex program

```
struct C3E2v_Output {
    float4 position : POSITION;
    float4 color : COLOR;
    float2 texCoord : TEXCOORD0;
};

C3E2v_Output C3E2v_varying(float2 position : POSITION,
                           float4 color : COLOR,
                           float2 texCoord : TEXCOORD0)
{
    C3E2v_Output OUT;
    OUT.position = float4(position, 0, 1);
    OUT.color = color;
    OUT.texCoord = texCoord;

    return OUT;
}
```

- OUT parameter values are passed to fragment shader
Example 1

- Result, assuming:
  - a fragment shader that just passes values through
  - OpenGL program
    ```
    glBegin(GL_TRIANGLES);
    glVertex( 0.8,  0.8); glColor(dark);
    glVertex(-0.8,  0.8); glColor(dark);
    glVertex( 0.0, -0.8); glColor(light);
    glEnd();
    ```
Example 2

- Fragment program, following example 1 vertex program

```c
struct C3E3f_Output {
  float4 color : COLOR;
};

C3E3f_Output C3E3f_texture(float2 texCoord : TEXCOORD0,
                           uniform sampler2D decal)
{
  C3E3f_Output OUT;
  OUT.color = tex2D(decal, texCoord);
  return OUT;
}
```

- Sampler2D is a texture object
  - other types exist: samplerRect, samplerCUBE, etc
Example 2

- Tex2D(decal, texCoord) performs a texture lookup
  - sampling, filtering, and interpolation depends on texture type and texture parameters
  - advanced fragment profiles allow sampling using texture coordinate sets from other texture units (dependent textures)
Example 2

- Result
Math Support

- A rich set of math operators and library functions
  - +-/*, sin, cos, floor, etc…
  - no bit-wise operators yet
- Full floating point operations
- Function overloading
  - for example, abs() function accepts float4, float2
Syntax

- **IN** keyword
  - call by value
  - parameter passing by value

- **OUT** keyword
  - indicates when the program returns
Example 3

**2D Twisting**

```c
struct C3E4_Output {
    float4 position : POSITION;
    float4 color : COLOR;
};

C3E4_Output C3E4_v_twist(float2 position : POSITION,
                          float4 color : COLOR,
                          uniform float twisting)
{
    C3E4_Output OUT;
    float angle = twisting * length(position);
    float cosLength, sinLength;
    sincos(angle, sinLength, cosLength);
    OUT.position[0] = cosLength * position[0] +
                      -sinLength * position[1];
    OUT.position[1] = sinLength * position[0] +
                      cosLength * position[1];
    OUT.position[2] = 0;
    OUT.position[3] = 1;
    OUT.color = color;
    return OUT;
}
```
Example 3

Result

finer meshes give better results
Example 4

Double Vision: vertex program

```c
void C3E5v_twoTextures(float2 position : POSITION,
                        float2 texCoord : TEXCOORD0,
                        out float4 oPosition : POSITION,
                        out float2 leftTexCoord : TEXCOORD0,
                        out float2 rightTexCoord : TEXCOORD1,
                        uniform float2 leftSeparation,
                        uniform float2 rightSeparation)
{
    oPosition = float4(position, 0, 1);
    leftTexCoord = texCoord + leftSeparation;
    rightTexCoord = texCoord + rightSeparation;
}
```

OUT is defined via semantics in the prototype
Example 4

- **Double Vision: fragment program #1**
  - samples the same texture (named *decal*) twice
- **lerp(a, b, weight): linear interpolation**
  - result = \((1\text{-weight}) \times a + \text{weight} \times b\)

```c
void C3E6f_twoTextures(float2 leftTexCoord : TEXCOORD0,
                        float2 rightTexCoord : TEXCOORD1,
                        out float4 color : COLOR,
                        uniform sampler2D decal)
{
    float4 leftColor = tex2D(decal, leftTexCoord);
    float4 rightColor = tex2D(decal, rightTexCoord);
    color = lerp(leftColor, rightColor, 0.5);
}
```
Example 4

- Result

Two Texture Samples for Each Fragment

leftSeparation = (-0.5, 0)
rightSeparation = (+0.5, 0)

Triangle with a Single Texture Coordinate Set

C3E5v_twoTextures & C3E6f_twoTextures
More Cg Examples
Green Sphere

- Uses a simple vertex program to color a sphere solid green.

- The fragment program just sets its color output to its color input (does nothing but pass the data along).
Color Sphere

- Same as green sphere, but the Cg Runtime is used to pass color in as a uniform parameter.
  
  ```
  float myColor[4] = { 1, 0, 0, 1 }; 
  cgGLSetParameter4fv(cgGetNamedParameter(vertexProgram, "iColor"), myColor);
  ```
Normal Vertex Sphere

- Uses a vertex program to shade a sphere with rgb components equal to the normal components.

- The normals are passed via a varying vertex parameter.
Normal Fragment Sphere

- Uses a fragment program to set the sphere color to the normalized interpolated vertex normal.
- The normals are passed to the vertex program via a varying parameters, and are passed to the fragment program via texture coordinates.
Texture Fragment Sphere

- Uses a fragment program to map a texture of the world to a sphere.

- The texture coordinates are passed to the vertex program via a varying parameter.

- The texture is passed directly to the fragment program as a "sampler".
“Plastic” Per-Vertex Shading

- Uses a vertex program to calculate standard "fixed function" lighting (aka plastic).

- Note the “faceted” nature of the shading, this will be addressed in the next example.
“Plastic” Per-Fragment Shading

- Uses a fragment program to perform standard "fixed-function" lighting (aka plastic).

- Note that the per-fragment calculations perform a much more accurate rendering (But slower)
Color Shaping using “lerp”

- Uses a fragment program to perform color shaping on the diffuse and specular components of the standard "fixed function" lighting.

- Color shaping is achieved by lerp'ing between two colors based on the diffuse and specular float values.

- `float3 specular = lerp(Ks0,Ks1,specularLight) * lightColor * specularLight;`
Normal Mapping

- Uses a fragment program to read normal and color information from texture maps and perform standard "fixed function" plastic lighting.
Homework2

- Please download cg toolkit

- Run the examples on your machine, make them working

- Study the examples for basic cg programming, together with our textbook