GPU Programming in Cg
GPU Programmable Graphics Pipeline

3D Apps

API commands

3D API: Direct3D

GPU cmd & data stream

Fixed Function Pipeline

NVidia GeForce FX

Programmable Vertex Shader

Programmable Fragment Shader

Transformed vertices

Transformed Fragments

GPU Frontend

Vtx index

Assembled polygons

Pixel location

Frame Buffer

Primitive Assembly

Rasterization & Interpolation

Raster Operations

API commands

GPU cmd & data stream

Source: Cg tutorial
Vertex and Pixel Shaders

• A vertex shader is a graphics function/program operating on vertex data
  – position (transformation), color (lighting), and texture.
  – replacing the fixed graphics transformation and viewing pipeline unit.

• A pixel (or fragment) shader is a programmable unit/program that operates on pixel data.
Shader Languages

- **Cg** (compatible with HLSL)
  - A graphics card supports Cg if the vendor includes the Cg compiler in the card’s driver.

- **Other options:**
  - HLSL most common post-DX 10.0
  - GLSL
  - Legacy DirectX shaders in assembly
  - Sh
  - OpenVidia (U of Toronto)
Cg Overview

• Cg is the High Level language from NVIDIA for programming GPUs, developed in close collaboration with Microsoft
  – Cg is 100% compatible with the HLSL in DirectX 9

• Cg has been offered to the OpenGL ARB as a proposal for the High Level Shading Language for future versions of OpenGL

• Cg enables a dramatic productivity increase for graphics development for:
  – Game developers
  – Artists & Shader writers
  – CAD and Visualization application application developers

“This is C for Graphics”
Graphics programs are written in Cg ...

... and compiled to ...

... low-level assembly code ...

... that runs on any GPU compatible with DirectX or OpenGL
Using the Cg Compiler

Application Development

Cg program source code

Cg Compiler

Shader program assembly code

Shader Compiler (nvasm.exe, psa.exe)

Shader Binary

Your Application

1) Load/bind program
2) Specify program parameters
3) Specify vertex inputs
4) Render

float d = dot(normalize(frag.N), normalize(frag.L));
if (d < 0)
  d = 0;
c = d*f4tex2D(t, frag.uv)*diffuse;
...
Compiling the programs

• Greater variation in GPU capabilities
  – Most processors don’t yet support branching
  – Vertex processors don’t support texture mapping
  – Some processors support additional data types

• Compiler can’t hide these differences
  – Least-common-denominator is too restrictive
  – Cg exposes differences via language profiles
    (list of capabilities and data types)

• The programs must be compiled to a certain profile
  – Input: A Cg program + profile to compile to
  – Output: Assembly language for the specified hardware
A Vertex Program

public class J6_1_Cg extends J1_5_Circle {
    CGcontext cgcontext;
    CGprogram vertexprog;

    // 1. Vertex profile: hardware specification/support
    static final int VERTEXPROFILE = CgGL.CG_PROFILE_ARBVP1;

    public void init(GLAutoDrawable glDrawable) {
        super.init(glDrawable);

        if (!CgGL.cgGLIsProfileSupported(VERTEXPROFILE)) {
            System.out.println("Profile not supported");
            System.exit(1);
        }
    }

    ...
}
public class J6_1_Cg extends J1_5_Circle {

    ...  

    // 2. Create Cg context for setting up the environment
    cgcontext=CgGL.cgCreateContext();

    // 3. Create vertex program from file with the profile
    CgGL.cgGLSetOptimalOptions(VERTEXPROFILE);
    vertexprog=CgGL.cgCreateProgramFromFile(cgcontext,
        CgGL.CG_SOURCE, "J6_1_VP.cg", VERTEXPROFILE, null, null);

    CgGL.cgGLLoadProgram(vertexprog);
}

...
A Vertex Program Conti.

public class J6_1_Cg extends J1_5_Circle {
...

    public void display(GLAutoDrawable drawable) {

        // 4. Enable the profile and binding the vertex program
        CgGL.cgGLEnableProfile(VERTEXPROFILE);
        CgGL.cgGLBindProgram(vertexprog);

        drawCircle(drawable);

        // 5. Disable the profile
        CgGL.cgGLDisableProfile(VERTEXPROFILE);
    }
...

public class J6_1_Cg extends J1_5_Circle {

    public void drawCircle(GLAutoDrawable drawable) {
        super.display(drawable);
    }

    public static void main(String[] args) {
        J6_1_Cg f = new J6_1_Cg();
        f.setTitle("JOGL J6_1_Cg");
        f.setSize(WIDTH, HEIGHT);
        f.setVisible(true);
    }
}
A Vertex Program Conti.

// Vertex Program: J6_1_VP.cg | update position and color

void main(
    in float4 iPosition : POSITION,
    in float4 iColor : COLOR,
    out float4 oPosition : POSITION,
    out float4 oColor : COLOR
) {
    oPosition.xyz = iPosition.xyz/1000; // division operator on vector
    oColor = iColor;
}

• “float4” represents a vector type of 4 components

• semantics represent actual hardware registers and connections.

• The “in” symbols, which are optional, represent input values (vertex position and color) from fixed graphics pipeline to the vertex shader (through registers).
A Vertex Program Conti.

// Vertex Program: J6_1_VP.cg   |   update position and color

void main(
    ...
) {
    oPosition.xyz = iPosition.xyz/1000; // division operator on vector
    oColor=iColor;
}

• The “out” symbols represent output values (vertex position and color).

• “oPosition.xyz” represents the first 3 components of “oPosition”. The 4th component is “oPosition.w”.

• the swizzling operation, vector1.xyz = vector2.yxz represents assignments of vector1.x = vector2.y, vector1.y = vector2.x, and vector1.z = vector2.z. Also, “iPosition.xyz/100” is a division operator on the vector. Cg comes with many standard library functions.
Transformation and Viewing

- Transformation, viewing, and projection on the vertices are now performed in the vertex shader.

- **MODELVIEW** and **PROJECTION** matrices need to be manually used in the vertex shader.

/* Cg Example: ModelviewProjection matrix */
public class J6_2_Cg extends J6_1_Cg {

    static CGparameter modelviewprojection;

...

 /* Cg Example: ModelviewProjection matrix */
public class J6_2_Cg extends J6_1_Cg {

    static CGparameter modelviewprojection;

    public void init(GLAutoDrawable glDrawable) {

        super.init(glDrawable);

        vertexprog=CgGL.cgCreateProgramFromFile(cgcontext,
CgGL.CG_SOURCE, "J6_2_VP.cg", VERTEXPROFILE, null, null);
        CgGL.cgGLLoadProgram(vertexprog);

        // modelview and projection matrix
        modelviewprojection=CgGL.cgGetNamedParameter(vertexprog,
"modelViewProjection");
    }
...
/* Cg Example: ModelviewProjection matrix */
public class J6_2_Cg extends J6_1_Cg {
    
    public void display(GLAutoDrawable drawable) {

        CgGL.cgGLEnableProfile(VERTEXPROFILE);
        CgGL.cgGLBindProgram(vertexprog);
        // retrieve the current modelview and projection matrix
        CgGL.cgGLSetStateMatrixParameter(modelviewprojection,
            CgGL.CG_GL_MODELVIEW_PROJECTION_MATRIX,
            CgGL.CG_GL_MATRIX_IDENTITY);

        drawCircle(drawable);

        CgGL.cgGLDisableProfile(VERTEXPROFILE);
    }
}
Transformation and Viewing Cont.

// J6_2_VP.cg Vertex Program: transformation and viewing
void main(
    float4 iPosition : POSITION,
    float4 iColor : COLOR,
out float4 position : POSITION,
out float4 color : COLOR,
uniform float4x4 : modelViewProjection
) {
    position = mul(modelViewProjection, iPosition);
    color = iColor;
}

• “uniform” is allocated by the JOGL program for the vertex program.

• The value can be changed by the JOGL program, but stays the same in the vertex program.

• “mul” will multiply the matrices together
A Fragment Program

/* J6_3_Cg: Setting up Fragment Program */

class J6_3_Cg extends J6_2_Cg {
    CGprogram fragmentprog;
    static final int FRAGMENTPROFILE=CgGL.CG_PROFILE_ARBFP1;
    
    public void init(GLAutoDrawable drawable) {
        super.init(drawable);
        if(!CgGL.cgGLIsProfileSupported(FRAGMENTPROFILE))
        {
            System.out.println("Fragment profile not supported");
            System.exit(1);
        }
        CgGL.cgGLSetOptimalOptions(FRAGMENTPROFILE);
        fragmentprog=CgGL.cgCreateProgramFromFile(cgcontext,
            CgGL.CG_SOURCE, "J6_3_FP.cg", FRAGMENTPROFILE, null, null);
        CgGL.cgGLLoadProgram(fragmentprog);
    }
    ...
}
A Fragment Program Cont.

/* J6_3_Cg: Setting up Fragment Program */
public class J6_3_Cg extends J6_2_Cg {
...

    public void display(GLAutoDrawable drawable) {
        CgGL.cgGLEnableProfile(VERTEXPROFILE);
        CgGL.cgGLBindProgram(vertexprog);
        CgGL.cgGLSetStateMatrixParameter(modelviewprojection,
            CgGL.CG_GL_MODELVIEW_PROJECTION_MATRIX,
            CgGL.CG_GL_MATRIX_IDENTITY);

        CgGL.cgGLEnableProfile(FRAGMENTPROFILE);
        CgGL.cgGLBindProgram(fragmentprog);

        drawCircle(drawable);

        CgGL.cgGLDisableProfile(VERTEXPROFILE);
        CgGL.cgGLDisableProfile(FRAGMENTPROFILE);
    }

}...


// J6_3_Cg: Setting up Fragment Program */
// J6_3_FP.cg Fragment Program: color manipulation
void main(
    float4 iColor : COLOR,
    out float4 color : COLOR
) {
    color.rgb = iColor.rgb / 2;
}

• The vertex colors of a primitive are interpolated along edges and then horizontal scan-lines for the pixels, so each fragment has an input color.
Uniform

• We can use “uniform” variables to pass information from the JOGL program to a Cg program.

• The following program generates random triangle colors in the JOGL program. The color is sent to the vertex program through a “uniform” variable.

/* Cg Example: uniform random colors */

public class J6_4_Cg extends J6_3_Cg {

    static CGparameter vertexColor;

    public void init(GLAutoDrawable glDrawable) {
        ...
        vertexColor = CgGL.cgGetNamedParameter(vertexprog, "vColor");
    }

    ...
}
rightness Cont.

/* Cg Example: uniform random colors */

public class J6_4_Cg extends J6_3_Cg {
    public void drawtriangle(float[] v1, float[] v2, float[] v3) {
        float color[] = new float[4];

        // generate a random color and set it to vertexColor
        color[0] = (float) Math.random();
        color[1] = (float) Math.random();
        color[3] = 0;
        CgGL.cgSetParameter4fv(vertexColor, color, 0);

        gl.glBegin(GL.GL_TRIANGLES);
        gl.glVertex3fv(v1, 0); gl.glVertex3fv(v2, 0);
        gl.glVertex3fv(v3, 0);
        gl.glEnd();
    }
}
/ J6_4_VP.cg Vertex Program: uniform vertex color

void main(
    float4 iPosition : POSITION,
    float4 iColor : COLOR,
out float4 position : POSITION,
out float4 color : COLOR,
uniform float4x4 modelViewProjection,
uniform float4 vColor
) {
    position = mul(modelViewProjection, iPosition);
    color=vColor;
}
Variable

- If “uniform” is not used, then a variable is either a semantic from the system or is defined explicitly through assignment

  - `float4 white = float4(1, 1, 1, 1).

  // J6_5_FP.cg Fragment Program: white color

  void main(
    float4 iColor : COLOR,
    out float4 color : COLOR,
    uniform float4 fColor
  ) {
    float4 white = float4(1,1,1,1);
    color =white;
  }

```
Per-Vertex Lighting

• Lighting in OpenGL
  – calculated after MODELVIEW transformation
  – light sources transformed by the MODELVIEW matrix.
• In Cg, we may calculate vertex lighting before or after MODELVIEW transformation.
• If we port an existing program with movable light source
  – transform the light source before sending it to the vertex shader.
  – send the matrix to the vertex shader to transform the light source.
• We have to send three matrices to the vertex shader
  – the MODELVIEW and PROJECTION matrix that transforms the vertex position for primitive assembly
  – the MODELVIEW matrix that transforms the vertex position for lighting calculations
  – the inverse transpose of the MODELVIEW matrix that transforms the vertex normal for lighting calculations
Per-Vertex Lighting (cont.)

static CGparameter
    modelviewprojection, // modelviewProjection matrix
    modelview, // modelview matrix
    inversetranspose, //inverse transpose of the modelview matrix

• There are many vertices. Therefore, it is better to calculate this transformation in the vertex shader:
  
  float4 vPosition = mul(modelView, iPosition);

• There are limited number of light sources, so calculate the transformation in the JOGL program. For example, :
  
  gl.glGetFloatv(GL.GL_MODELVIEW_MATRIX, currM, 0);
  sphereC[0] = currM[12];
  sphereC[1] = currM[13];
  sphereC[2] = currM[14];
  CgGL.cgSetParameter3fv(myLightPosition, sphereC, 0);
Per-Vertex Lighting (cont.)

• Whenever we retrieve the current matrix for vertex transformation, we should retrieve the inverse transpose for normal transformation:

  CgGL.cgGLSetStateMatrixParameter(modelview,  
  CgGL.CG_GL_MODELVIEW_MATRIX,  
  CgGL.CG_GL_MATRIX_IDENTITY);

  CgGL.cgGLSetStateMatrixParameter(inversetranspose,  
  CgGL.CG_GL_MODELVIEW_MATRIX,  
  CgGL.CG_GL_MATRIX_INVERSE_TRANSPOSE);

  CgGL.cgGLSetStateMatrixParameter(modelviewprojection,  
  CgGL.CG_GL_MODELVIEW_PROJECTION_MATRIX,  
  CgGL.CG_GL_MATRIX_IDENTITY);
Per-Vertex Lighting (cont.)

• In the Jogl program
  static CGparameter
    myLa, //light source ambient
    myLd, //light source diffuse
    myLs, //light source specular
    myLightPosition, // light source position
    myEyePosition,
    myMe, // material emission
    myMa, // material ambient
    myMd, // material diffuse
    myMs, // material specular
    myShininess; // material shininess

• In the vertx program
  – the vertex position is transformed to the eye space by the MODELVIEW matrix:
    float4 vPosition = mul(modelView, iPosition);
    float3 P = vPosition.xyz;
  – The light source direction is from the current vertex to the light source position:
    float3 L = normalize(lightPosition - P);
  – The emission and ambient components:
    float3 Ie = Me;
    float3 Ia = La*Ma;
Per-Vertex Lighting (cont.)

- In the vertex program...
  - The diffuse component is as follows. Again, “max” and “dot” are Cg standard library functions:
    
    \[
    \text{float cosNL} = \max(\text{dot}(N, L), 0);
    \]
    
    \[
    \text{float3 Id} = \text{Md} \times \text{Ld} \times \text{cosNL};
    \]

  - For the specular component, the viewpoint direction is from the viewpoint (eyePosition) to the vertex position:
    
    \[
    \text{float3 V} = \text{normalize(eyePosition} - \text{P});
    \]
    
    \[
    \text{float3 H} = \text{normalize(L} + \text{V});
    \]
    
    \[
    \text{float cosNH} = \max(\text{dot}(N, H), 0);
    \]
    
    \[
    \text{if (cosNL}==0) \text{cosNH} = 0; // condition in Cg
    \]
    
    \[
    \text{float3 Is} = \text{Ms} \times \text{Ls} \times \text{pow(cosNH, shininess)};
    \]

  - Finally, we have the single lighting model in the vertex shader:
    
    \[
    \text{oColor.xyz} = \text{Ie} + \text{Ia} + \text{Id} + \text{Is};
    \]
    
    \[
    \text{oPosition} = \text{mul(modelViewProjection, iPosition)};
    \]
Per-Fragment Lighting

• Need the vertex position and normal in the eye space interpolated across the primitive

• This is achieved using semantics TEXCOORD0 and TEXCOORD1
  
  – The output in the vertex shader to TEXCOORD0 and TEXCOORD1 are passed on to the pixel shader as input TEXCOORD0 and TEXCOORD1, respectively.

  – we calculate vertex position and normal in the vertex shader, but we send them to the pixel shader through TEXCOORD0 and TEXCOORD1 for actual lighting calculation
Per-Fragment Lighting (cont.)

// J7_2_VP.cg Vertex Program: fragment lighting

void main(
    float4 iPosition : POSITION,
    float4 iNormal : NORMAL,
    out float4 oPosition : POSITION,
    out float4 vPosition : TEXCOORD0,
    out float4 vNormal : TEXCOORD1,

    uniform float4x4 modelView,
    uniform float4x4 modelViewProjection,
    uniform float4x4 inverseTranspose
)
{
    vPosition = mul(modelView, iPosition);
    vNormal = mul(inverseTranspose, iNormal);
    vNormal.xyz = normalize(vNormal.xyz);
    oPosition = mul(modelViewProjection, iPosition);
}
Per-Fragment Lighting (cont.)

• Since the lighting is calculated in the pixel shader, we should send all the lighting parameters to it:
  
  myLa = CgGL.cgGetNamedParameter(fragmentprog, "La");
  myLd = CgGL.cgGetNamedParameter(fragmentprog, "Ld");
  myLs = CgGL.cgGetNamedParameter(fragmentprog, "Ls");
  myLightPosition = CgGL.cgGetNamedParameter(fragmentprog, "lightPosition");
  myEyePosition = CgGL.cgGetNamedParameter(fragmentprog, "eyePosition");
  myMe = CgGL.cgGetNamedParameter(fragmentprog, "Me");
  myMa = CgGL.cgGetNamedParameter(fragmentprog, "Ma");
  myMd = CgGL.cgGetNamedParameter(fragmentprog, "Md");
  myMs = CgGL.cgGetNamedParameter(fragmentprog, "Ms");
  myShininess = CgGL.cgGetNamedParameter(fragmentprog, "shininess");
// J7_2_VP.cg Fragment Program: fragment lighting

void main(float4 iPosition : TEXCOORD0,
          float4 iNormal : TEXCOORD1,
out float4 oColor : COLOR,
        uniform float3 La,
        ...
) {

    //interpolated position and normal values
    float3 P = iPosition.xyz;
    float3 N = normalize(iNormal.xyz);
    float3 L = normalize(lightPosition - P);

    //calculate emission and ambient components
    float3 Ie = Me;
    float3 Ia = La*Ma;

    // calculate diffuse component
    float cosNL = max(dot(N, L), 0);
    float3 Id = Md * Ld * cosNL;

    // calculate specular component
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float cosNH = max(dot(N, H), 0);
    if (cosNL==0) cosNH = 0;
    float3 Is = Ms * Ls * pow(cosNH, shininess);

    oColor.xyz = Ie + Ia + Id + Is;
}
Per-Fragment Texture Mapping

- A vertex’s texture coordinates are sent to the vertex shader through semantics TEXCOORD0.
  - This is default similar to the vertex position and color
  - texture coordinates are fixed values at the vertices.
- We can then pass the texture coordinates to the pixel shader through a TEXCOORD semantics
  - which will interpolate the texture coordinates for the pixels (fragments) across the corresponding primitive
- In the JOGL program, the current texture object (through glBindTexture) needs to be sent to the Pixel Shader for texel retrieval.
Per-Fragment Texture Mapping Cont.

// J7_3_VP.cg Vertex Program: fragment texture mapping
void main(
    float4 iPosition : POSITION,
    float4 iNormal : NORMAL,
    float2 iTexCoord : TEXCOORD0,// input texture coord.
    out float4 oPosition : POSITION,
    out float4 vPosition : TEXCOORD0,
    out float4 vNormal : TEXCOORD1,
    out float2 oTexCoord : TEXCOORD2,// output to pixel shader
    uniform float4x4 modelView,
    uniform float4x4 modelViewProjection,
    uniform float4x4 inverseTranspose
) {
    vPosition = mul(modelView, iPosition);
    vNormal = mul(inverseTranspose, iNormal);
    vNormal.xyz = normalize(vNormal.xyz);
    oTexCoord = iTexCoord;
    oPosition = mul(modelViewProjection, iPosition);
}
Per-Fragment Texture Mapping Cont.

- The current texture object in the Jogl program

  ```java
  static CGparameter imgtexure; // texture object name

  // texture object name for Pixel Shader
  imgtexure = CgGL.cgGetNamedParameter(fragmentprog, "imgTexure");

  gl.glBindTexture(GL.GL_TEXTURE_2D, EARTH_TEX[0]);

  CgGL.cgGLSetTextureParameter(imgtexure, EARTH_TEX[0]);
  CgGL.cgGLEnableTextureParameter(imgtexure);
  ```

- In the Fragment program, texture is retrieved from library function tex2D:

  ```utf-8
  uniform sampler2D imgTexure : TEX0

  // retrieve texture from imgTexture at iTexCoord
  float4 texColor = tex2D(imgTexure, iTexCoord);
  ```
Per-Fragment Texture Mapping Cont.

• Here “imgTexture” a “uniform sampler2D” type with semantics TEX0: uniform sampler2D imgTexure TEX0,

• 1D, 3D, CUBE, and other type of built-in sampling application types exist.

• **Texture and Lighting Blending**
  Cg library function has a linear interpolation function “lerp”,

  \[
  \text{oColor} = \text{lerp}(\text{texColor}, \text{oColor}, a);
  \]

  is equivalent to:
  \[
  \text{oColor} = (1 - a)\times\text{texColor} + a\times\text{oColor};
  \]
Per-Fragment Bump Mapping

• First, we need to initialize the bump map as a texture:

```java
void initTexture() {
    // initialize bumpmap texture obj
    gl.glGenTextures(1, IntBuffer.wrap(NORMAL_TEX));
    gl.glBindTexture(GL.GL_TEXTURE_2D, NORMAL_TEX[0]);
    gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER,
                       GL.GL_LINEAR);
    gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER,
                       GL.GL_LINEAR);
    readImage("NORMAL.jpg");

    gl.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGB8,
                    imgW, imgH, 0, GL.GL_BGR, GL.GL_UNSIGNED_BYTE,
                    ByteBuffer.wrap(img));
    super.initTexture();
}
```
**Per-Fragment Bump Mapping Cont.**

- Then, we need to bind the bump map texture name:

  ```
  CgGL.cgGLSetTextureParameter(normalmap, NORMAL_TEX[0]);
  CgGL.cgGLEnableTextureParameter(normalmap);
  ```

- where “normalmap” is a CGparameter:

  ```
  static CGparameter normalmap; // bump map object name

  // texture object name for Pixel Shader
  normalmap = CgGL.cgGetNamedParameter(fragmentprog, "normalMap");
  ```

- In the fragment program, bump map is retrieved from library function tex2D:

  ```
  // retrieve bump map vector from normalMap at iTexCoord
  float4 texColor1 = tex2D(normalMap, iTexCoord);

  float3 N = texColor1.xzy*2 - 1;
  ```
• **Normal Calculations**
  
  • First, we define an arbitrary vector $T$:
    
    ```
    float4 T = float4(iTexCoord.x, iTexCoord.y, 0, 0);
    float4 N = mul(inverseTranspose, iNormal);
    ```

  • Therefore, by two cross products we can find $TNB$ as follows:
    
    ```
    nNormal = N.xyz;
    tNormal = T.xyz;
    bNormal = cross(tNormal, nNormal);
    tNormal = cross(nNormal, bNormal);
    tNormal = normalize(tNormal);
    nNormal = normalize(nNormal);
    bNormal = normalize(bNormal);
    ```
/ J7_4_VP.cg Vertex Program: bump mapping
void main(
    float4 iPosition : POSITION,
    float4 iNormal : NORMAL,
    float2 iTexCoord : TEXCOORD0,
out float4 oPosition : POSITION,
out float2 oTexCoord : TEXCOORD0,
out float4 vPosition : TEXCOORD1,
out float3 nNormal : TEXCOORD2,
out float3 tNormal : TEXCOORD3,
out float3 bNormal : TEXCOORD4,
uniform float4x4 modelView,
uniform float4x4 modelViewProjection,
uniform float4x4 inverseTranspose
) {
    vPosition = mul(modelView, iPosition);
    float4 T = float4(iTexCoord.x, iTexCoord.y, 0, 0);
    float4 N = mul(inverseTranspose, iNormal);
    ...
Per-Fragment Bump Mapping Cont.

// J7_4_VP.cg Vertex Program: bump mapping
void main(
    ...
) {
    vPosition = mul(modelView, iPosition);
    float4 T = float4(iTexCoord.x, iTexCoord.y, 0, 0);
    float4 N = mul(inverseTranspose, iNormal);

    nNormal = N.xyz;
    tNormal = T.xyz;
    bNormal = cross(tNormal, nNormal);
    tNormal = cross(nNormal, bNormal);
    tNormal = normalize(tNormal);
    nNormal = normalize(nNormal);
    bNormal = normalize(bNormal);

    oTexCoord = iTexCoord;
    oPosition = mul(modelViewProjection, iPosition);
}
// J7_4_FP.cg Fragment Program: fragment bump mapping
void main(
    float2 iTexCoord : TEXCOORD0,
    float4 iPosition : TEXCOORD1,
    float3 nNormal : TEXCOORD2,
    float3 tNormal : TEXCOORD3,
    float3 bNormal : TEXCOORD4,
out float4 oColor : COLOR,
uniform sampler2D imgTexture : TEX0,
uniform sampler2D normalMap : TEX0,
    uniform float3 La,
    uniform float3 Ld,
    uniform float3 Ls,
    uniform float3 lightPosition,
    uniform float3 eyePosition,
    uniform float3 Me,
    uniform float3 Ma,
    uniform float3 Md,
    uniform float3 Ms,
    uniform float shininess
) {
    ...
}
Per-Fragment Bump Mapping Cont.

// J7_4_FP.cg Fragment Program: fragment bump mapping
void main(
    ...
) {
    // retrieve bump map vector at iTexCoord
    float4 texColor1 = tex2D(normalMap, iTexCoord);
    float4 texColor2 = tex2D(imgTexture, iTexCoord);

    // retrieve pixel position and normal
    float3 N = texColor1.xzy*2 - 1;
    float3 P = iPosition.xyz;

    // transform light source direction to tangent space
    float3 Lg = normalize(lightPosition - P);
    float3 L = float3(dot(tNormal, Lg), dot(nNormal, Lg), dot(bNormal, Lg)) ;

    ...
}
// J7_4_FP.cg Fragment Program: fragment bump mapping
void main(
    ...
) {
    ...

    // calculate emission and ambient components
    float3 le = Me;    float3 la = La*Ma;

    // calculate diffuse component
    float cosNL = max(dot(N, L), 0);
    float3 Id = Md * Ld * cosNL;

    // calculate specular component
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float cosNH = max(dot(N, H), 0);
    if (cosNL==0) cosNH = 0;
    float3 Is = Ms * Ls * pow(cosNH, shininess);

    oColor.xyz = le + la + Id + Is;
    oColor = lerp(oColor, texColor2, 0.5);