Concepts and language fundamentals

Real-time Graphical Shader Programming with Cg (HLSL)

Presentation Outline

- Objectives
- Resources
- Shaders at a glance
- GPU pipeline
- History
- Cg/HLSL
- Programming
- CgFX

Objectives

- Understand purposes/uses of shaders
- Be able to discuss shaders intelligently
- Be able to program/use shaders
- Will not cover techniques

Resources

- Reading Material
  - *Cg Tutorial from NVIDIA
- Software
  - *Cg Toolkit (compilers, etc.)
  - *Cg Tutorial (editable examples from Cg Tutorial)
  - SDK9, SDK10.5 (detailed samples)
  - FX Composer (content authoring)
- NVIDIA developer's site has more

Shaders at a Glance
What is a Shader?

- Program typically used for graphical effects
- Runs on GPU
- Controls shape, appearance, and motion of objects
- Per-Object or full-screen
- Responsible for handling most OpenGL and DirectX effects
- May be used outside of games

Shaders at a Glance Example

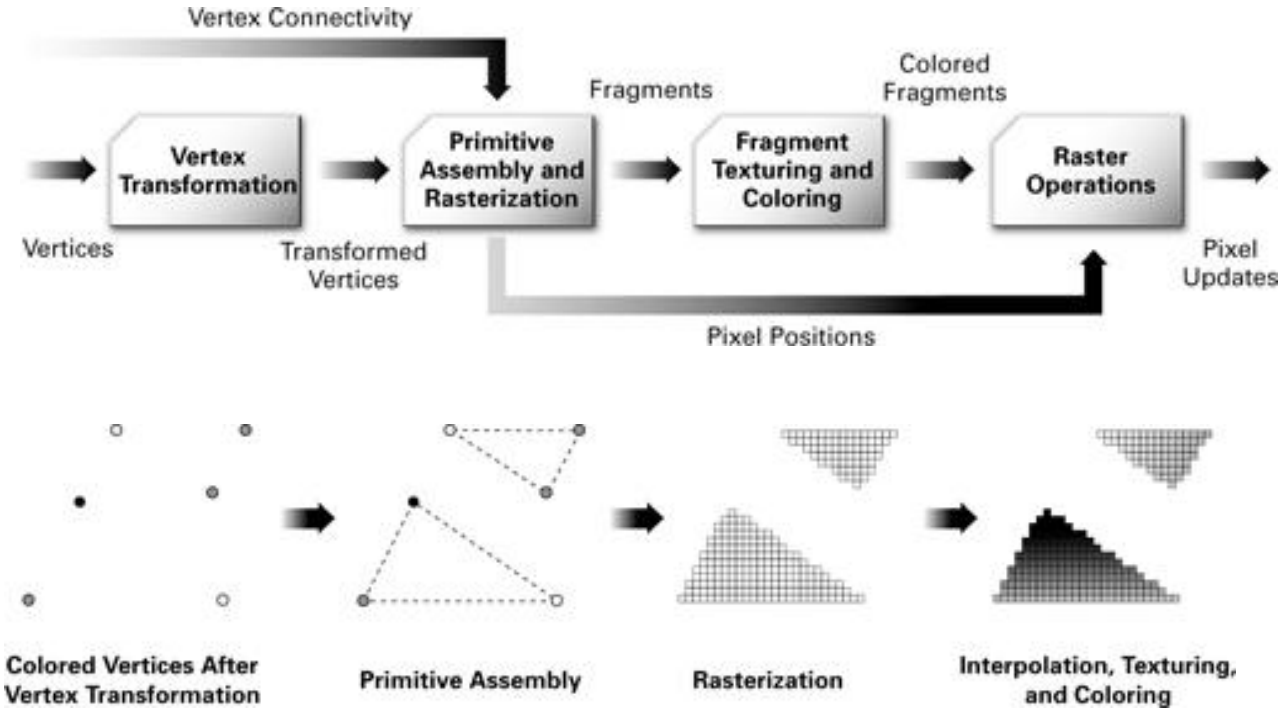
- Lighting (basic)
- Human Head (advanced)

Shaders at a Glance
Types of Shaders

- Vertex
  - Operates on vertices
- Geometry
  - Requires DirectX10
  - Not covered here
- Fragment (pixel)
  - Operates on fragments (think pixels)

Modern GPU Pipeline

- From chapter 1 of the Cg Tutorial



History
Graphics Cards

- Video Graphics Array (VGA)
  - IBM introduced in 1987
  - "dumb" frame buffer
- Pre-GPU by SGI
  - Vertex transformation and texturing
- Graphics Processing Unit (GPU)
  - NVIDIA introduced term in late 1990s

History Graphics Cards

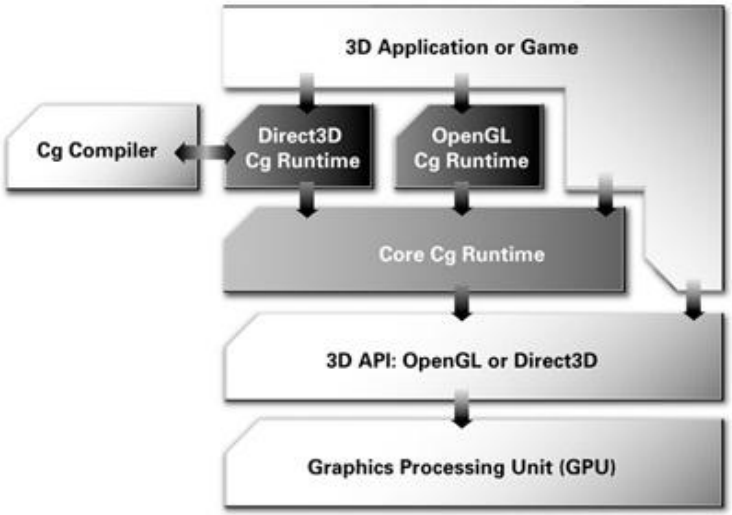
- 1st Generation (<= 1998)
  - Only basic texturing
  - Implement DirectX 6
- 2nd Generation (1999-2000)
  - Configurable
  - Adds vertex transformation and lighting
  - DirectX 7
- 3rd Generation (2001)
  - Programmable (barely)
  - Fragment "shaders" in assembly
  - DirectX 8
- 4th Generation (>= 2002)
  - Fully programmable with high level languages
  - DirectX 9

Cg/HLSL

- Developed in collaboration
  - Same language, different implementations
  - Borrows ideas from RenderMan
- C for Graphics (Cg)
  - NVIDIA's implementation
  - OpenGL or Direct3D
- High-Level Shading Language (HLSL)
  - Microsoft's implementation
  - Direct3D

Programming Compilation

- Compiler part of Cg runtime library
  - Cg runtime routines prefixed with "cg"
- Dynamic compilation preferred
  - Allows specific optimizations
- Compilation parameters include
  - Shader program filename
  - Entry function name (e.g. "main")
  - Profile name
- Errors
  - Conventional
  - Profile-dependent
    - Program not supported by specified profile



Programming Profiles

- Defines the mapping from source to native code
- Depends on
  - Shader program type
    - Vertex or fragment
  - Language features used
  - Graphics API used
  - GPU capabilities
- Basic profiles are most portable
- Advanced profiles allow more features
  - May produce more efficient code

Programming Simple Example

- Vertex program

```
struct outType {
  float4 position : POSITION;
  float4 color : COLOR;
};

outType shadeGreen(float2 position : POSITION) {
  outType OUT;
  OUT.position = float4(position, 0, 1);
  OUT.color = float4(0, 1, 0, 1); // RGBA green
  return OUT;
}
```

Programming Control Flow

- Vertex program -> fragment program
  - Don't need both*
- Entry function
  - Specified when compiling shader
  - Can use internal functions

Programming Data Types

- Everything is floating-point

```
float scalar;
half ;
double ;
```

- Everything is vector-based (even scalars)
- Packed arrays (more efficient than regular arrays)

```
float4 vec4;
float4x4 myMatrix;
```

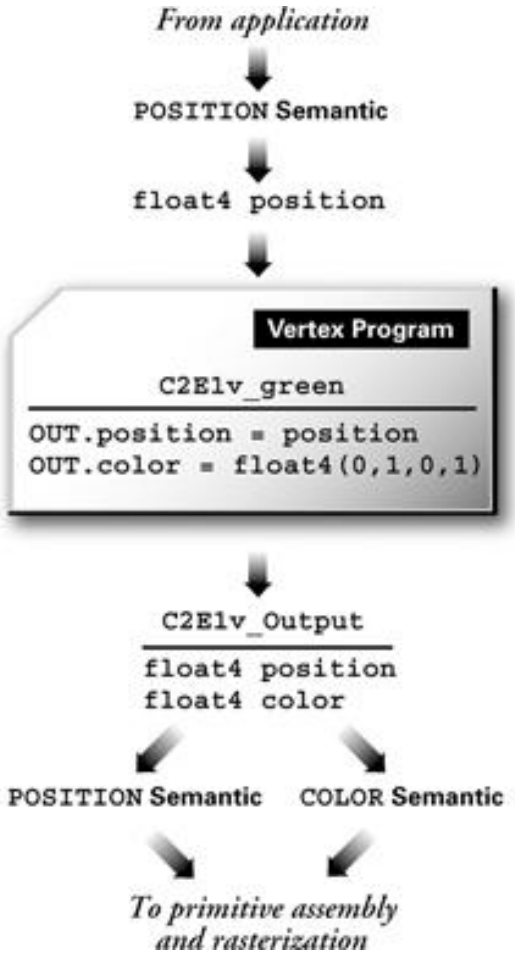
- Operations optimized but random accesses not

```
scalar = vec4.z;       // Efficient
scalar = vec4[3];      // Efficient
scalar = data[index]; // Inefficient or unsupported
```

Programming Semantics

- Link between pipeline and Cg program
- Use existing or make your own
- Ignored by internal functions
- Input/Output semantics are different
  - Represent same concept
  - Different stages of graphics pipeline
- Examples

```
float4 position : POSITION
float4 color : COLOR
float2 texCoord : TEXCOORD0
```



Programming Type Qualifiers

- **`const`**
  - As in C/C++
- **`in`**
  - Assumed
- **`out`**
  - Copy-out
- **`inout`**
  - Copy-in-copy-out
- **`uniform`**
  - Parameter comes from external source
  - Similar to required command line arguments

Programming Swizzling

- Swizzling Vectors

```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);
float2 vec2 = vec1.yx;     // vec2 = (-2.0, 4.0)
```

- Smearing

```
float scalar = vec1.w;     // scalar = 3.0
float3 vec3 = scalar.xxx; // vec3 = (3.0, 3.0,
  3.0)
```

- Swizzling Matrices

```
float4x4 myMatrix;         // assign somewhere
float4 vec4 = myMatrix._m00_m11_m22_m33;
vec4 = myMatrix[0];
```

Programming Write Masking

```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);
float2 vec2 = float2(-2.0, 4.0);
vec1.xw = vec2;   // vec1 = (-2.0, -2.0, 5.0, 4.0)
```

- Note
  - Can use .xyzw or .rgba, but cannot mix the two in a single swizzle or mask

Programming Transformations

- Some linear algebra
  - Rotation matrix
  - Order is important

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- Need to know
  - Shaders operate in clip space
  - Manual transformation required

Programming Transformations Example

```
void justTransform(
      float4 position : POSITION,
      out float4 oPosition : POSITION,
      uniform float4x4 modelViewProj)
{
   // Transform position from object space to clip
   space oPosition = mul(modelViewProj, position);
}
```

Programming Projectile Example

```
void projectile(
    float4 pInitial : POSITION,
    uniform float4 vInitial,
    uniform float tInitial,  // animation start time
    uniform float globalTime,// current time
    uniform float4 acceleration,
    uniform float4x4 modelViewProj,
    out float4 pFinal : POSITION)
{
    float t = globalTime - tInitial;

    pFinal = pInitial +
             vInitial * t +
             0.5 * acceleration * t * t;

    pFinal = mul(modelViewProj, pFinal);
}
```

CgFX

- Unified file format
  - Superset of Cg
- Multiple implementations and passes
- Additional configuration options
- Adds annotations
  - Allowable ranges for parameters