Graphics Hardware

Computer Graphics
COMP 770 (236)
Spring 2007

Instructor: Brandon Lloyd
From last time...

- Texture coordinates
- Uses of texture maps
  - reflectance and other surface parameters
  - lighting
  - geometry
- Solid textures
Graphics from a system’s perspective

- Graphics operations most frequently executed on a co-processor called a Graphics Processing Unit (GPU)
- Dedicated buses between the “host” CPU and the GPU
  - AGP, PCI Express
- Separate GPU memory
  - Framebuffer, textures, etc.
- Shared memory with CPU
OpenGL 1.4 - Graphics Past

- Fixed-function graphics pipeline
  - every step neatly planned
- PHILOSOPHY: Performance > Flexibility
- Extended by committee
- Why process anything other than polygons or the occasional pixel?

A fragment is a potential
OpenGL 2.0 - Graphics Today

- Programmable processing units
  - Programmable vertex and fragment processors
  - (Exposes what was always there beneath the covers)
- Texture memory – general-purpose data storage
A GPU block diagram

- GeForce 6 Series
- Massive parallelism
  * pipelining
  * multiple data paths
- Mix of programmable and hardwired function blocks
- Simple inter-processor connectivity
- High-bandwidth memory interfaces
Vertex processor capabilities

- Lighting, Material and Geometry flexibility
- **Vertex programs replace the following parts of the pipeline:**
  - Vertex & Normal transformation
  - Normalization and rescaling
  - Per-Vertex Lighting Calculations
  - Color application
  - Texture coordinate generation & transformation
- **The vertex program does NOT replace:**
  - Perspective divide and viewport (NDC) mapping
  - Clipping
  - Backface culling
  - Primitive assembly (Triangle setup, edge equations, etc.)
Vertex processor Inputs & Outputs

- Vertex “shader” is supplied with a number of parameters
  - Vertex parameters, OpenGL state, user supplied parameters
- Results written into prearranged locations (registers) that are “understood” by later processing steps

**User-Defined Uniform Variables**
- eyePosition, lightPosition, modelScaleFactor, etc.

**Standard OpenGL attributes**
- glColor, glNormal
- glVertex, glMultiTexCoord

**User-Defined Attributes**
- User-Defined Uniform Variables
- User-Defined Uniform Variables
- User-Defined Uniform Variables

**Standard OpenGL State**
- ModelViewMatrix, glLightSource[0..n]
- glFogColor, glFrontMaterial, etc.

**Vertex Processor**
- Vertex & texture coords, Vertex color
- Model coordinates, Normals, hVector, toEyeVector, etc.
Fragment processor capabilities

- Flexibility for texturing and per-pixel operations
- Fragment programs replace the following parts of the OpenGL pipeline:
  - Operations on interpolated values
  - Texture access
  - Texture application (modulate, add)
  - Fog (color, depth)
  - Color sums (blends, mattes)
  - Pixel zoom
  - Scale and bias
  - Color table lookup
  - Convolution
  - Color matrix
- The Fragment shader does NOT replace:
  - Scan Conversion
  - Coverage
  - Scissor
  - Alpha test
  - Stencil test
  - Logical ops
  - Plane masking
  - Histogram
  - Pixel packing and unpacking
  - Stipple
  - Depth test
  - Alpha blending
  - Dithering
  - Z-buffer replacement test
Fragment processor Inputs & Outputs

- Fragment “shader” is supplied with a number of parameters
  - fragment parameters, OpenGL state, user supplied parameters
- Results written into prearranged locations (registers) that are “understood” by later processing steps

User-Defined Uniform Variables
eyePosition, lightPosition, modelScaleFactor, epsilon, etc.

Standard Rasterizer attributes
color (r, g, b, a), depth (z), textureCoordinates

User-Defined Attributes
Normals, modelCoord, density, etc.

Fragment Processor

TextureMemory
Textures, Tables, TempStorage

Standard OpenGL variables
FragmentColor, FragmentDepth
The first Vertex and Fragment programs were written in low-level, H/W-specific assembly languages
- specific capabilities (eg. floating point only in Vertex shaders, fixed-point only in Fragment shaders)

Trend is toward Higher-Level languages
- GeForce 8800 has unified shaders (same capabilities for both Vertex and Fragment shaders)
Historical: GeForce 3 Vertex Processor

- In the beginning, resources were limited
- Difficult to do anything, even at the assembly level
- Useful macros for
  - Vector-scalar mult
  - Vector-vector add
  - Dot-product
  - Normalize

became the programming method of choice
GPU/CPU Differences

- Early GPUs offered no branching support
- Conditional operations instead
  
  \[
  \text{If (regA} < 0) \\
  \text{regB} = \text{regC}
  \]

- No general indirect access to memory (i.e. lookup tables, textures, etc.)
- Limited Arrays (uniform parameters)
- Fixed vector sizes (2, 3 & 4)
**Recent GPUs (GeForce 6)**

- 512 total instructions
  64K executed per primitive
- Independent execution (MIMD)
- Branching
- Subroutine calls
- Flexible per-vertex processing
- General purpose vector registers 4-element (x,y,z,w)
- Vertex Cache
- Texture accesses
Recent fragment processors

- 512 total instructions
  64K executed per primitive
- Coupled execution (SIMD)
- Branching
- Subroutine calls
- Flexible per-fragment processing
- General purpose vector registers
  4-element (r,g,b,a)
- Streamed Datapaths
- Texture accesses

Excerpted from GPU Gems 2
Copyright NVIDIA Corp. 2005
Fragment unit performance

- 4 ops & 2 Units
- Same operation on all 4-args on 2 units
- Or 2-different ops on 2-args on 2 units per clock
- RGBA or RGBA
- 16-units per chip
- 128 = 16 x 8 ops per clock

Excerpted from GPU Gems 2
Copyright NVIDIA Corp. 2005
GPU Assembly Example:

FRC R2.y, C11.w;
ADD R3.x, C11.w, -R2.y;
MOV H4.y, R2.y;
ADD H4.x, -H4.y, C4.w;
MUL R3.xy, R3.xyww, C11.xyww;
ADD R3.xy, R3.xyww, C11.z;
TEX H5, R3, TEX2, 2D;
ADD R3.x, R3.x, C11.x;
TEX H6, R3, TEX2, 2D;

- GPU programs were quirky and non-portable
High-level languages to the rescue

*Cg* High-level language example: (*Cg* = “C for graphics”)

```c
float4 main(    float2 detailCoords : TEXCOORD0,
                float2 bumpCoords: TEXCOORD1,
                float3 lightVector : COLOR0,
                uniform float3 ambientColor,
                uniform sampler2D detailTexture : TEXUNIT0,
                uniform sampler2D bumpTexture : TEXUNIT1): COLOR

{    float3 detailColor = tex2D(detailTexture, detailCoords).rgb;

    float3 lightVectorFinal = 2.0 * (lightVector.rgb - 0.5);
    float3 bumpNormalVectorFinal = 2.0 * (tex2D(bumpTexture, bumpCoords).rgb - 0.5);

    float diffuse = dot(bumpNormalVectorFinal, lightVectorFinal);

    return float4(diffuse * detailColor + ambientColor, 1.0);
}
```

- Easier to read and maintain (old-lesson relearned), but still quirky…
- Symbolic “named” variables (allocated by compiler)
- Portability between implementations
- Reuse pieces
Explosion of GPU HLLs

- **Stanford Shading Language**
  - (Predecessor to Cg) C/Renderman-like

- **Cg**
  - Separate programming environment
  - Compiled and linked into application

- **GLSL (OpenGL Shading Language)**
  - Embedded into application and compiled on the fly

- **HLSL - DirectX pixel-shading language**
  - Integrated into application programming environment
Shading language uses

- Lots of versatility

Subsurface Scattering  NPR Renders  Fire Effects  Refraction

Ray Tracing  Solid Textures  Ambient Occlusion  Cloth Simulation
HLLs are still hard to use

- All of the old problems associated with managing projects with multiple parts resurface
  - Separate “host” program
  - Separate “Vertex” program
  - Separate “Fragment” program
  - Multiple “shaders” per application
  - Want incremental “tweaks” of shaders to propagate to successors
  - Due to small program size—must manage loading and running multipass shaders

- What’s the solution?
  - Smart syntax aware editors
  - Project Managements scripts (Makefiles)
  - Wrappers, Linking loaders, (OOP??)
Enter the GPU IDE
FX Composer & RenderMonkey

- **Integrated Shader Development Environment**
  - Interactive Preview window-- lets you see the impact of your shader changes immediately
  - Supports HLSL (RM also supports GLSL)
  - Separate editor windows for vertex and fragment shading code
  - Support generation of artwork (textures, color palettes, MIPmaps)
  - Built-in host application that allows loading geometry
  - Built-in disassembler
  - Provides error checking and limited debugging

- **Free download at**
GPGPU

General Processing on Graphics Processing Units (GPGPU)

- map more general computation to the GPU to take advantage of the massive amounts of raw processing power
Graphics hardware future

- **GeForce 8800**
  - unified shader architecture
  - general scalar processors (integer and bitwise ops)
Graphics hardware future

- Convergence between CPUs and GPUs
  - CPUs going multi-core
    - many simpler, lower power cores
  - GPUs becoming more and more programmable
    - CUDA, PeakStream, etc.
- Massive multi-threading to hide latency
- Heterogeneous collection of processors
  - Cell
  - ATI Fusion
- More complex memory models
  - less hardware caching
  - more reliance on the programmer
Next time

- Programmable shaders