Modern GPU Architectures
CPSC 314

Rendering Pipeline

So far:
- Have discussed rendering pipeline as a specific set of stages with fixed functionality

Modern graphics hardware is more flexible:
- Programmable “vertex shaders” replace several geometry processing stages
- Programmable “fragment/pixel shaders” replace texture mapping stage
- Hardware with these features now called “Graphics Processing Unit” (GPU)

Modified Pipeline

Vertex shader
- Replaces model/view, lighting, and perspective
- Have to implement these yourself
- But can also implement much more

Fragment/pixel shader
- Replaces texture mapping
- Fragment shader must do texturing
- But can do other things

Vertex Shader Motivation

Hardware “transform&lighting:
- I.e. hardware geometry processing
- Was mandated by need for higher performance in the late 90s
- Previously, geometry processing was done on CPU, except for very high end machines
- Downside: now limited functionality due to fixed function hardware

Vertex Shaders

Programmability required for more complicated effects
- The tasks that come before transformation vary widely
- Putting every possible lighting equation in hardware is impractical
- Implementing programmable hardware has advantages over CPU implementations
  - Better performance due to massively parallel implementations
  - Lower bandwidth requirements (geometry can be cached on GPU)
Vertex Program Properties

Run for every vertex, independently
- Access to all per-vertex properties
  - Position, color, normal, texture coords, other custom properties
- Access to read/write registers for temporary results
- Value is reset for every vertex
- I.e. cannot pass information from one vertex to the next
- Access to read-only registers
  - Global variables, like light position, transformation matrices
- Write output to a specific register for the resulting color

Vertex Shaders/Programs

Concept:
- Programmable pipeline stage
  - Floating-point operations on 4-vectors
    - Points, vectors, and colors!
- Replace all of
  - Model/View Transformation
  - Lighting
  - Perspective projection

Vertex Program Properties – Instruction Set

Arithmetic Operations on 4-vectors:
- ADD, MUL, MAD, MIN, MAX, DP3, DP4

Operations on Scalars
- RCP (1/x), RSQ (1/√x), EXP, LOG

Specialty Instructions
- DST (distance: computes length of vector)
- LIT (quadratic falloff term for lighting)

Later generation:
- Loops and conditional jumps

IO for Vertex Shaders (Circa 2001)

- Newer hardware has more instructions, more memory

- Register that are constant for all vertices:
  - position, color, texture coordinates…
- Registers that are constant for all vertices (changes are expensive):
  - Matrices, light position and color, …
- Temporary registers
- Output registers for position, color, tex coords…

Vertex Shaders/Programs

Concept:
- A little assembly-style program is executed on every individual vertex
- It sees:
  - Vertex attributes that change per vertex:
    - position, color, texture coordinates…
  - Registers that are constant for all vertices (changes are expensive):
    - Matrices, light position and color, …
  - Temporary registers
  - Output registers for position, color, tex coords…

Vertex Programs – Applications

What can they be used for?
- Can implement all of the stages they replace, but can allocate resources more dynamically
  - E.g. transforming a vector by a matrix requires 4 x 4 dot products
  - Enough memory for 24 matrices
  - Can arbitrarily deform objects
  - Procedural freeform deformations
  - Lots of other applications
    - Shading
    - Refraction
    - …
**Vertex Programming Example**

*Example (from Stephen Cheney)*
- Morph between a cube and sphere while doing lighting with a directional light source (gray output)
- Cube position and normal in attributes (input) 0.1
- Sphere position and normal in attributes 2,3
- Blend factor in attribute 15
- Inverse transpose model/view matrix in constants 12-14
  - Used to transform normal vectors into eye space
- Composite matrix is in 4-7
  - Used to convert from object to homogeneous screen space
- Light dir in 20, half-angle vector in 22, specular power, ambient, diffuse and specular coefficients all in 21

**Vertex Program Example**

```
Example
```

```
// blend normal and position
// use cvw_modelview(x) = cvw(v) = cvw(v)[0] x cvw(v)[1] x cvw(v)[2] x cvw(v)
MOV r3, v[1] ;
MUL r5, r3, r3 ;
ADD r6, v[1], -R3 ;
ADD r7, v[2], -R5 ;
MOV r8, v[2] ;
MOV r9, v[3] ;
MOV r10, v[3] ;
MOV r11, v[3] ;
# transform normal to eye space
DPS r12, x, r8, c[12] ;
DPS r13, y, r9, c[13] ;
DPS r14, z, r10, c[14] ;
# normalize normal
DPS r15, w, r9, R9 ;
PS0 r15, r15, r15 ;
MUL R5, r9, R9, R9 ;
# apply lighting and output color
DPS r16, x, r9, c[20] ;
DPS r17, y, R9, c[22] ;
MOV r18, x, c[21] ;
LT r3, r0 ;
DPS a[COLOR], c[21], R1 ;
```

**Skinning**

*Example was one case of general problem:*
- Want to have natural looking joints on human and animal limbs
- Requires deforming geometry, e.g.
  - Single triangle mesh modeling both upper and lower arm
  - If arm is bent, upper and lower arm remain more or less in the same shape, but transition zone at elbow joint needs to deform

**Skinning**

*Approach:*
- Multiple transformation matrices
  - There is more than one model/view matrix stack, e.g.
    - one for model/view matrix for lower arm, and
    - one for model/view matrix for upper arm
  - Every vertex is transformed by both matrices
    - Yields 2 different transformed vertex positions!
  - Use per-vertex blending weights to interpolate between the two positions

### Arm Example:

- M1: matrix for upper arm
- M2: matrix for lower arm

#### Transition zone
- weight for M1 between 0.1
- weight for M2 between 0.1

---

© Wolfgang Heidrich

© Wolfgang Heidrich

© Wolfgang Heidrich

© Wolfgang Heidrich
Skinning

In general:
- Many different matrices make sense!
  - EA facial animations: up to 70 different matrices ("bones")
  - Hardware supported:
    - Number of transformations limited by available registers and max. instruction count of vertex programs
    - But dozens are possible today

GeForce FX Fragment/Pixel Program Examples

Source: David Kirk/NVIDIA

Fragment Shader Motivation

The idea of per-fragment shaders have been around for a long time
- Renderman is the best example, but not at all real time

In a traditional pipeline, the only major per-pixel operation is texture mapping
- All lighting, etc. is done in the vertex processing, before primitive assembly and rasterization
- In fact, a fragment is only screen position, color, and texture coordinates

What kind of shading interpolation does this restrict you to?

Fragment Shader Generic Structure

Fragment Shader Functionality

At a minimum, we want to be able to do Phong interpolation
- How do you get normal vector info?
- How do you get the light?
- How do you get the specular color?
- How do you get the world position?
Shading Languages

- Programming shading hardware is still a difficult process
  - Akin to writing assembly language programs
- Shading languages and accompanying compilers allow users to write shaders in high level languages
- Two examples: Microsoft’s HLSL (part of DirectX 9) and Nvidia’s Cg (compatible with HLSL)
- Renderman is the ultimate example, but it’s not real time

Cg

Cg is a high-level language developed by NVIDIA

- It looks like C or C++
- Actually a language and a runtime environment
  - Can compile ahead of time, or compile on the fly
  - Why compile on the fly?
- What it can do is tightly tied to the hardware
  - How does it know which hardware, and how to use it?

Pixel Program Example

```c
void CGF_fragmentLighting(float4 position : POSITION, 
float4 normal : NORMAL, 
out float4 output : COLOR, 
// inner shader execution state 
shared float4 color[] = tex2D(colorTexture, texcoord));

// inner shader execution state 
shared float4 color[] = tex2D(colorTexture, texcoord));
```

Vertex Program Example

```c
cgsmc_fragmentLighting(float4 position : POSITION, 
float4 normal : NORMAL, 
out float4 output : COLOR, 
// inner shader execution state 
shared float4 color[] = tex2D(colorTexture, texcoord));
```

Cg Runtime

- There is a sequence of commands to get your Cg program onto the hardware
**Bump/Normal Mapping**

**Normal Mapping Approach:**
- Directly encode the normal into the texture map
  - \((R,G,B) = (x,y,z)\), appropriately scaled
- Then only need to perform illumination computation
- Interpolate world-space light and viewing direction from the vertices of the primitive
  - Can be computed for every vertex in a vertex shader
  - Get interpolated automatically for each pixel
- **In the fragment shader:**
  - Transform normal into world coordinates
  - Evaluate the lighting model

---

**Latest Developments: Geometry Shaders**

**“Direct X 10” Hardware**
- Geometry shaders
- ...
Geometry Shader Example

Generalized displacement maps

Displacement Mapping (Direct3D 10)

Source: Glassenberg/Microsoft

Single Pass Render-To-Cubemap

Source: Glassenberg/Microsoft

Single Pass Render-To-Cubemap

Source: Glassenberg/Microsoft

Coming Up...

Thursday:
- Shadows

Tuesday:
- Color