Modern GPU Architectures

CPSC 314

The Rendering Pipeline
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

---

**IO for Vertex Shaders** *(Circa 2001)*

- Newer hardware has more instructions, more memory
**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 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 – 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

---

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 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

```
# blend normal and position
# v = αv₁ + (1-α)v₂ = α(v₁-v₂) + v₂
MOV R3, v[3];
MOV R5, v[2];
ADD R8, v[1], -R3;
ADD R6, v[0], -R5;
MAD R8, v[15].x, R8, R3
MAD R6, v[15].x, R6, R5;

# transform normal to eye space
DP3 R9.x, R8, c[12];
DP3 R9.y, R8, c[13];
DP3 R9.z, R8, c[14];

# transform position and output
DP4 o[HPOS].x, R6, c[4];
DP4 o[HPOS].y, R6, c[5];
DP4 o[HPOS].z, R6, c[6];
DP4 o[HPOS].w, R6, c[7];

# normalize normal
DP3 R9.w, R9, R9;
RSQ R9.w, R9.w;
MUL R9, R9.w, R9;

# apply lighting and output color
DP3 R0.x, R9, c[20];
DP3 R0.y, R9, c[22];
MOV R0.zw, c[21];
LIT R1, R0;
DP3 o[COL0], 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

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
Skinning

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

- Upper arm:
  - weight for M1=1
  - weight for M2=0

- Transition zone:
  - weight for M1 between 0..1
  - weight for M2 between 0..1

- Lower arm:
  - weight for M1=0
  - weight for M2=1

Example by NVIDIA

© 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 tex-coords

*What kind of shading interpolation does this restrict you to?*

---

**Fragment Shader Generic Structure**

[Diagram showing the generic structure of a fragment shader]

*Figure 6.20. Generalized pixel shader. Variants in the pixel shader language primarily affect the way texture address instructions work, as a result storage can be stored, and whether the z-depth can be modified and output.*
**Fragment Shaders**

- *Fragment shaders* operate on fragments in place of the texturing hardware
  - After rasterization, before any fragment tests or blending
- Input: The fragment, with screen position, depth, color, and a set of texture coordinates
- Access to textures and some constant data and registers
- Compute RGBA values for the fragment, and depth
  - Can also “kill “a fragment, that is throw it away
- Two types of fragment shaders: register combiners (GeForce4) and fully programmable (GeForceFX, Radeon 9700)

**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?

---

Vertex Program Example

```c
void CSE2v_fragmentLighting(float4 position : POSITION,
                            float3 normal : NORMAL,
                            out float4 oPosition : POSITION,
                            out float3 objectPos : TEXCOORD0,
                            out float3 cNormal : TEXCOORD1,
                            uniform float4x4 modelViewProj)
{
    oPosition = mul(modelViewProj, position);
    objectPos = position.xyz;
    cNormal = normal;
}
```

© Wolfgang Heidrich
Pixel Program Example

```c
void CSHF_basictlight(float4 position, float3 normal)
{
    out float4 color; COLOR.

    uniform float3 globalambient,
    uniform float3 lightColor,
    uniform float3 lightPosition,
    uniform float3 eyePosition,
    uniform float3 Rs, uniform float3 Re, uniform float3 R0,
    uniform float3 Kd, uniform float3 Kr

    ( // Compute the diffuse term
        float P = position.xyz;
        float N = normalize(normal);
        float diffuse = max(dot(N, Le), 0.0);
        float diffuse = N * lightColor * diffuse;

        // Compute the specular term
        float3 H = normalize(eyePosition - P);
        float3 Rs = normalize(eyePosition - P);
        float3 R0 = normalize(eyePosition - P);
        float3 Kr = normalize(eyePosition - P);
        float3 specular = max(pow(dot(N, H), 2.0), 0.0) * lightColor * diffuse;
        float3 color = diffuse + ambient + diffuse * specular;
        color.w = 1;
    }
```

© Wolfgang Heinrich

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

Bump Mapping

**Examples**
Latest Developments: Geometry Shaders

"Direct X 10" Hardware

- Geometry shaders
- ...

Full control over the whole triangle

- All-GPU Material Systems
- Better materials
  - Hi-quality interpolation and derivatives
  - Wrinkle models
  - Cartoon and falloff effects

Geometry/data amplification

- Fur/Fins
- Procedural geometry/detailing
- All-GPU Particle Systems
- Data visualization techniques
- Wide lines and strokes
- ...

Source: Glassenberg/Microsoft
Geometry Shader Example

**Shadow volume generation**

Source: Glassenberg/Microsoft

---

Geometry Shader Example

**Generalized displacement maps**

*Normal mapping (Direct3D)*

Source: Glassenberg/Microsoft
Geometry Shader Example

Generalized displacement maps

Displacement Mapping (Direct3D 10)

Source: Glassenberg/Microsoft

Single Pass Render-To-Cubemap

Geometry Shader

Source: Glassenberg/Microsoft
Single Pass Render-To-Cubemap

Source: Glassenberg/Microsoft

Coming Up...

**Thursday:**
- Shadows

**Tuesday:**
- Color