The GPU Revolution: Programmable Graphics Hardware

David Luebke
University of Virginia

First a Shameless Plug
Real-Time Rendering

- Consider CS 446: Real-time rendering
  - Tue-Thu 12:30-1:45 MEC 214
  - Interesting topics:
    - scene graphs, visibility and occlusion, shadows, terrains,
    - level of detail, non-photorealistic rendering, Cg and shaders, balancing the rendering pipeline, advanced texturing, parallel rendering, image-based rendering, interactive ray tracing, virtual reality...
  - Semester-long group project: full-fledged game engine
  - Intense programming effort, lots of fun

Modern Graphics Pipeline

- Past couple of years have seen a true revolution in graphics hardware
  - Programmability
  - Precision

Modern Graphics Pipeline

- Past couple of years have seen a true revolution in graphics hardware
  - Programmability
    - Old: “fixed-function pipeline” with all stages hard-coded
    - New: Major stages of pipeline programmable
  - Precision
Modern Graphics Pipeline

- Past couple of years have seen a true revolution in graphics hardware
  - Programmability
    - Old: “fixed-function pipeline” with all stages hard-coded
    - New: Major stages of pipeline programmable
  - Precision
    - Old: most input and all output at 8 bits of RGBA
    - New: true floating-point available per channel, from 16 to 32-bit IEEE floats, throughout pipeline
    - Why do we want this extra precision?

Classic Rendering Pipeline
(as presented in CS 445)

Modeling
Transformation
Viewing
Transformation
Projection
Transformation
Lighting
3D Geometric Primitives
Image
Clipping
Scan Conversion

Transform into 3D world coordinate system
Transform into 3D camera coordinate system
Draw pixels (includes texturing, hidden surface, ...)

What’s wrong with this model
(for an OpenGL system)?
  - Model/view transforms combined
    - Then how does lighting work?
  - Really “vertices” not “primitives”
    - Making this the vertex pipeline
      - There’s a lot going on in the “scan conversion” stage:
        - Primitive assembly
        - Rasterization
        - Texture mapping
        - Per-pixel lighting
        - Visibility (z-buffer)
      - We refer to these collectively as the pixel or fragment pipeline

Modern OpenGL Pipeline

Application
Vertex Processor
Assembly & Rasterization
Pixel Processor
Video Memory (Textures)

- Note:
  - Vertex processor does all transform and lighting
  - Pipe widths vary
    - Intra-GPU pipes wider than CPU-GPU pipe
    - Thin GPU-to-GPU pipe
  - Many caches and FIFOs not shown
  - Soon: render-to-vertex-array

- Here’s what’s cool:
  - Can now program vertex processor!
  - Can now program pixel processor!
Outline

- Outline for rest of talk:
  - Specific details on NV3X architecture
  - Programming the GPU: Cg
  - Examples
  - Coming soon

GPU Programming Model
From an NVIDIA talk

32-bit IEEE floating-point throughout pipeline

- Framebuffer
- Textures
- Fragment processor
- Vertex processor
- Interpolants

Hardware supports several other data types

- Fragment processor also supports:
  - 16-bit “half” floating point
  - 12-bit fixed point
  - These may be faster than 32-bit on some HW

- Framebuffer/textures also support:
  - Large variety of fixed-point formats
  - E.g., classical 8-bit per component
  - These formats use less memory bandwidth than FP32
**Vertex processor capabilities**

- 4-vector FP32 operations, as in GeForce3/4
- True data-dependent control flow
  - Conditional branch instruction
  - Subroutine calls, up to 4 deep
  - Jump table (for switch statements)
- Condition codes
- New arithmetic instructions (e.g. COS)
- User clip-plane support

**Vertex processor has high resource limits**

- 256 instructions per program (effectively much higher with branching)
- 16 temporary 4-vector registers
- 256 “uniform” parameter registers
- 2 address registers (4-vector)
- 6 clip-distance outputs

**Fragment processor has clean instruction set**

- General and orthogonal instructions
- Much better than previous generation
- Same syntax as vertex processor:
  ```
  MUL R0, R1.xyz, R2.yxw;
  ```
- Full set of arithmetic instructions: RCP, RSQ, COS, EXP, ...

**Fragment processor has flexible texture mapping**

- Texture reads are just another instruction (TEX, TXP, or TXD)
- Allows computed texture coordinates, nested to arbitrary depth
- Allows multiple uses of a single texture unit
- Optional LOD control - specify filter extent
- Think of it as...
  A memory-read instruction, with optional user-controlled filtering
Additional fragment processor capabilities

- Read access to window-space position
- Read/write access to fragment Z
- Built-in derivative instructions
  - Partial derivatives w.r.t. screen-space x or y
  - Useful for anti-aliasing
- Conditional fragment-kill instruction
- FP32, FP16, and fixed-point data

Fragment processor limitations

- No branching
  - But, can do a lot with condition codes
- No indexed reads from registers
  - Use texture reads instead
- No memory writes

Fragment processor has high resource limits

- 1024 instructions
- 512 constants or uniform parameters
  - Each constant counts as one instruction
- 16 texture units
  - Reuse as many times as desired
- 8 FP32 x 4 perspective-correct inputs
- 128-bit framebuffer “color” output
  (use as 4 x FP32, 8 x FP16, etc...)

Programming in assembly is painful

- Easier to read and modify
- Cross-platform
- Combine pieces
- etc.

```
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;
...```
Cg - “C for Graphics”

- Cg is a high-level GPU programming language
- Designed by NVIDIA and Microsoft
- Competes with the (quite similar) GL Shading Language, a.k.a GLslang

Design goals for Cg

- Enable algorithms to be expressed...
  - Clearly, and
  - Efficiently
- Provide interface continuity
  - Focus on DX9-generation HW and beyond
  - But provide support for DX8-class HW too
  - Support both OpenGL and Direct3D
- Allow easy, incremental adoption

Easy adoption for applications

- Avoid owning the application’s data
  - No scene graph
  - No buffering of vertex data
- Compiler sits on top of existing APIs
  - User can examine assembly-code output
  - Can compile either at run time, or at application-development time
- Allow partial adoption
  - e.g. Use Cg vertex program with assembly fragment program
- Support current hardware

Some points in the design space

- CPU languages
  - C - close to the hardware; general purpose
  - C++, Java, lisp - require memory management
  - RenderMan - specialized for shading
- Real-time shading languages
  - Stanford shading language
  - Creative Labs shading language
Some points in the design space

- CPU languages
  - C - close to the hardware; general purpose
  - C++, Java, Lisp - require memory management
  - RenderMan - specialized for shading

- Real-time shading languages
  - Stanford shading language
  - Creative Labs shading language

Design strategy

- Start with C (and a bit of C++)
  - Minimizes number of decisions
  - Gives you known mistakes instead of unknown ones
- Allow subsetting of the language
- Add features desired for GPU's
  - To support GPU programming model
  - To enable high performance
- Tweak to make it fit together well

How are current GPU’s different from CPU?

1. GPU is a stream processor
   - Multiple programmable processing units
   - Connected by data flows

Cg uses separate vertex and fragment programs

![Diagram of Cg pipeline]
Cg programs have two kinds of inputs

- Varying inputs (streaming data)
  - e.g. normal vector - comes with each vertex
  - This is the default kind of input
- Uniform inputs (a.k.a. graphics state)
  - e.g. modelview matrix
- Note: Outputs are always varying

```c
vout MyVertexProgram(float4 normal,
                      uniform float4x4 modelview)
{
  ...
}
```

Two ways to bind VP outputs to FP inputs

a) Let compiler do it
- Define a single structure
- Use it for vertex-program output
- Use it for fragment-program input

```c
struct vout {
  float4 color;
  float4 texcoord;
  ...
};
```

b) Do it yourself
- Specify register bindings for VP outputs
- Specify register bindings for FP inputs
- May introduce HW dependence
- Necessary for mixing Cg with assembly

```c
struct vout {
  float4 color : TEX3 ;
  float4 texcoord : TEX5;
  ...
};
```

Some inputs and outputs are special

- e.g. the position output from vert prog
  - This output drives the rasterizer
  - It must be marked

```c
struct vout {
  float4 color;
  float4 texcoord;
  float4 position : HPOS;
};
```
How are current GPU’s different from CPU?

2. Greater variation in basic 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
   - We expose differences via language profiles (list of capabilities and data types)
   - Over time, profiles will converge

3. Optimized for 4-vector arithmetic
   - Useful for graphics - colors, vectors, texcoords
   - Easy way to get high performance/cost

   - C philosophy says: expose these HW data types
   - Cg has vector data types and operations e.g. float2, float3, float4
   - Makes it obvious how to get high performance
   - Cg also has matrix data types e.g. float3x3, float3x4, float4x4

Some vector operations

```c
// Clamp components of 3-vector to [minval,maxval] range
//
float3 clamp(float3 a, float minval, float maxval) {
  a = (a < minval.xxx) ? minval.xxx : a;
  a = (a > maxval.xxx) ? maxval.xxx : a;
  return a;
}
```

Cg has arrays too

- Declared just as in C
- But, arrays are distinct from built-in vector types: float4 != float[4]
- Language profiles may restrict array usage

```c
evout MyVertexProgram( float3 lightcolor[10], ...
```
How are current GPU’s different from CPU?

4. No support for pointers
   - Arrays are first-class data types in Cg
5. No integer data type
   - Cg adds “bool” data type for boolean operations
   - This change isn’t obvious except when declaring vars

Cg basic data types

- All profiles:
  - float
  - bool
- All profiles with texture lookups:
  - sampler1D, sampler2D, sampler3D, samplerCUBE
- NV_fragment_program profile:
  - half  -- half-precision float
  - fixed  -- fixed point [-2,2]

Other Cg capabilities

- Function overloading
- Function parameters are value/result
  - Use “out” modifier to declare return value

```c
void foo (float a, out float b) {
    b = a;
}
```

```c
if (a > b)
    discard;
```

Cg Built-in functions

- Texture mapping (in fragment profiles)
- Math
  - Dot product
  - Matrix multiply
  - Sin/cos/etc.
  - Normalize
- Misc
  - Partial derivative (when supported)
- See spec for more details
Cg Example - part 1

// In:
//     eye_space position = TEX7
//     eye space T = (TEX4.x, TEX5.x, TEX6.x)  denormalized
//     eye space B = (TEX4.y, TEX5.y, TEX6.y)  denormalized
//     eye space N = (TEX4.z, TEX5.z, TEX6.z)  denormalized

fragout frag program main(vf30 In) {

    float m = 30;                        // power
    float3 hiCol = float3( 1.0, 0.1, 0.1 );  // lit color
    float3 lowCol = float3( 0.3, 0.0, 0.0 );  // dark color
    float3 specCol = float3( 1.0, 1.0, 1.0 );  // specular color

    // Get eye-space eye vector.
    float3 e = normalize( -In.TEX7.xyz );

    // Get eye-space normal vector.
    float3 n = normalize( float3(In.TEX4.z, In.TEX5.z, In.TEX6.z ) );

    float  edgeMask = (dot(e, n) > 0.4) ? 1 : 0;
    float3 lpos = float3(3,3,3);
    float3 l = normalize(lpos - In.TEX7.xyz);
    float3 h = normalize(l + e);
    float specMask = (pow(dot(h, n), m) > 0.5) ? 1 : 0;
    float hiMask = (dot(l, n) > 0.4) ? 1 : 0;
    float3 ocol1 = edgeMask *
        (lerp(lowCol, hiCol, hiMask) + (specMask * specCol));

    fragout O;
    O.COL = float4(ocol1.x, ocol1.y, ocol1.z, 1);
    return O;
}

Cg Example - part 2

float edgeMask = (dot(e, n) > 0.4) ? 1 : 0;
float3 lpos = float3(3,3,3);
float3 l = normalize(lpos - In.TEX7.xyz);
float3 h = normalize(l + e);
float specMask = (pow(dot(h, n), m) > 0.5) ? 1 : 0;
float hiMask = (dot(l, n) > 0.4) ? 1 : 0;
float3 ocol1 = edgeMask *
    (lerp(lowCol, hiCol, hiMask) + (specMask * specCol));

fragout O;
O.COL = float4(ocol1.x, ocol1.y, ocol1.z, 1);
return O;

What does this shader look like?

Toon Shader

- This is a simple a “toon shader” designed to give a cartoonish look to the geometry

New vector operators

- Swizzle - replicate/rearrange elements
  a = b.xxx;
- Write mask - selectively over-write
  a.w = 1.0;
- Vector constructor builds vector
  a = float4(1.0, 0.0, 0.0, 1.0);
Change to constant-typing mechanism

- In C, it's easy to accidentally use high precision
  ```c
  half x, y;
  x = y * 2.0; // Double-precision multiply!
  ```
- Not in Cg
  ```c
  x = y * 2.0; // Half-precision multiply
  ```
- Unless you want to
  ```c
  x = y * 2.0f; // Float-precision multiply
  ```

Dot product, Matrix multiply

- Dot product
  ```c
  dot(v1, v2); // returns a scalar
  ```
- Matrix multiplications:
  ```c
  - matrix-vector: mul(M, v); // returns a vector
  - vector-matrix: mul(v, M); // returns a vector
  - matrix-matrix: mul(M, N); // returns a matrix
  ```

Coming Soon...

- Future hardware and drivers will be exposing even more programmability
- Current-generation chips: NV3X, R3XX
  - The first fully-programmable parts
  - More or less the same feature set
    - ATI R300: only 24-bit precision, no 16-bit support, shorter programs, less flexible dependent texturing, better performance
    - ATI R350: Includes an F-buffer which stores and replays fragments in rasterization order
  - Not currently exposed, though

Coming Soon...

- Next-generation chips: NV40, R400
  - Still under wraps, but ps3.0 gives us an idea
  - Expect:
    - At least some branching in fragment program
    - Much longer programs (virtualized to multipass)
    - Flexible memory: render-to-vertex array, etc.
    - Faster readbacks (with PCI-express)
  - Don’t expect:
    - Precision higher than 32-bit
Closing Comments

- Programmable GPUs have literally changed the world of graphics
- “Wheel of reincarnation” is turning
  → Game and graphics developers must understand fundamentals!
- You can do interesting non-graphics stuff on a GPU... (demo)
- GPUs are changing much faster than CPUs...
  → These are exciting times in graphics