Programming shaders & GPUs

Christian Miller
CS 354 - Fall 2011
Fixed-function vs. Programmable

• Up until 2001, graphics cards implemented the whole pipeline for you
  - Fixed functionality but configurable through the graphics API and extensions
• Starting with GeForce 3, you could start programming chunks of the pipeline yourself
  - Written as shaders, first in assembly, then in more usable languages
Shaders today

• Basically all graphics hardware for the last 5 years has shipped with shader support
• The pipeline is still implemented in hardware, but several parts are totally programmable
• DirectX and OpenGL’s recommended modes are completely shader based, OpenGL ES requires it
• If you still want the old functionality, implement it yourself in a shader!
Shader pipeline

- Green = programmable
- Yellow = configurable
- Blue = fixed
**Vertex shader**

- Takes one submitted vertex, generates one output vertex
- Usually used to:
  - Multiply each submitted vertex by projection and modelview matrices
  - Perform per-vertex lighting and texcoord calculations
  - Apply shape warping effects
Geometry shader

• Takes incoming primitives and may generate additional vertices and primitives

• Generally used to:
  • Perform tesselation, surface evaluation, or subdivision
  • Dynamically handle level-of-detail
**Pixel shader**

- Takes a fragment and interpolated vertex attribute data, outputs framebuffer values for merging

- Usually used to:
  - Evaluate per-pixel lighting models
  - Do texture lookups and blending
Things not handled in shaders

- Perspective divide
- Clipping
- Vertex attribute interpolation
- Fragment generation
- Merging fragments into the framebuffer
Compatibility

• Vertex and pixel shaders have been around since DirectX 8 and OpenGL 2.0
  • Support is basically guaranteed
• Geometry shaders are newer, in DirectX 10 and OpenGL 3.2
  • Support for these is still not universal
• Other types of shaders have shown up in DirectX 11 and OpenGL 4.0, but support is very spotty
Programming shaders

• Step 1: Write your shaders in GLSL

• Step 2: In your C program, create a shader object for each shader, load up the code, and have the driver compile it for you

• Step 3: Create a GL program context, attach the compiled shaders to it, and have the driver link it

• Step 4: Let the driver know which program you want to use, then start rendering
Writing shaders

- GLSL is a C-like language used to write shaders
- Just write everything in text files, like regular code
- Vector math is built in (exactly like GLM)
- Can have several functions spread across files
- We’ll cover shader language 1.20 here
  - Old (OpenGL 2.1, 2006), but very common
Simple example

```glsl
#include <glad/glad.h>

void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```glsl
#include <glad/glad.h>

void main()
{
    gl_FragColor = vec4(0.4, 0.4, 0.8, 1.0);
}
```
Inputs / outputs

- Each shader has a set of inputs and outputs named beginning with “gl_”
- These are set up automatically for you
- Some of them must be written in order to have a valid shader
- Can also make user-defined I/Os
I/O TYPES

• const: Constants known at shader compile time
• uniform: Constants known at render time
• attribute: Submitted vertex attributes (input)
• varying: Written as output by vertex shaders, then interpolated and used as input by pixel shaders
Built-in uniforms

- Made available to all shaders
- gl_ModelViewMatrix, gl_ProjectionMatrix, gl_NormalMatrix, gl_ModelViewMatrixInverse, etc...
**Vertex shader I/Os**

- **Inputs (attribute):** gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord\(n\)
- **Outputs (varying):** gl_Position, gl_FrontColor, gl_BackColor, gl_TexCoord\([n]\)
Pixel shader I/Os

• Input (varying): gl_FragCoord, gl_FrontFacing, gl_Color, gl_TexCoord[n]
• Output: gl_FragColor, gl_FragDepth
Another example

```glsl
#version 120

varying vec3 c;

void main()
{
    c = gl_Normal;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```glsl
#version 120

const float alpha = 0.8;
varying vec3 c;

void main()
{
    gl_FragColor = vec4(c.x, c.y, c.z, alpha);
}
```
Loading and using shaders

- Create shader objects in OpenGL, bind them together into a program, then use it for rendering
Creating shader objects

• Use glCreateShader() with the type of shader to create a new number for it

• Then, use glShaderSource() to attach your source to that shader

• Finally, use glCompileShader() to compile it
  • Use glGetShaderInfoLog() to get a log explaining the errors if compilation failed
Creating program objects

- Use `glCreateProgram()` to get a number for a new program object
- Use `glAttachShader()` to attach the compiled vertex and fragment shaders to it
- `glLinkProgram()` will link them together
  - Use `glGetProgramInfoLog()` to get errors
- Finally, `glUseProgram()` will tell GL to use that set of shaders for rendering
Submitting data

- Once compiled, both uniforms and attributes get numbers assigned to them
- Use `glGetUniformLocation()` and `glGetAttribLocation()` to get those numbers
- `glUniform()`, `glUniformMatrix()` and so forth will load in uniform data
- `glVertexAttrib()` will submit per-vertex data
More examples...

- The website for the OpenGL orange book has lots of examples: http://3dshaders.com/
GPU programming

• GPUs have become common and general enough that you can use them outside of graphics

• They’re massively powerful, but only if the work can be executed completely in parallel
  - Synchronous parallel, to be exact...

• Very recently, APIs have become available to allow you to write arbitrary code for GPUs
Two GPGPU languages

- CUDA: Nvidia’s language, arrived in 2007
  - Extended C++ syntax, API is custom
  - Works only on Nvidia hardware
- OpenCL: Multi-vendor, proposed in late 2008
  - C99 syntax, API mimics OpenGL
  - Works across GPUs, CPUs, etc...
How GPGPU works

• You write a kernel (like a shader)
  • This has the code you actually want to run
• Your host program creates a context, loads up the program and its data (along with how it’s distributed), and tells it to execute asynchronously
• Host can go off and do other things, and gets a notification when the computation is done
Good performance

- Writing GPGPU code (and shaders) that perform well is all about knowing GPU architecture

- That’s next lecture...