Shaders in OpenGL

Fixed versus Programmable

Demographics of a GPU

Anatomy of a Shader Program

Shader Component Programs

Why use Shaders?

• What we’ve done until now: used a “hardwired” pipeline
  • Produces limited effects
  • Effects look the same (e.g. Lighting!)
  • Gamers want unique look and feel
  • Multi-texturing w/ combiners helps, but still limited
  • Less interoperable, less portable

• Current graphics practice: use programmable GPU
  • A shader is a small program that runs on a “GPU”
    • Written in a high-level shader language (HLSL, Cg, GLSL)
    • OpenGL natively supports GLSL Language (others by extension)
    • Allows greater rendering flexibility
    • Achieve visual effects that would be difficult or impossible w/ fixed
    • Offload highly parallel work from CPU for greater performance

Demographics of a GPU

• “Graphics Processing Unit”
• Highly Parallel SIMD Architecture
  • Up to 10^6 concurrent threads!
  • Hardware thread scheduling
  • In-order issue
• NVidia 8800GTX (high-end G80 core)
  • 16 Stream Multiprocessors
  • Each SM contains 8 unified streaming processors – 128 in total
• NVidia GTX280 (high-end GT200 core)
  • 24 Stream Multiprocessors
  • Each SM contains 8 unified streaming processors – 240 in total
  • ~1.5 billion transistors

Anatomy of a Shader Program

• A Shader is a program written in textual form in some high
  level Shader Language
• Programs tend to have these components:
  • Global variables
    • Built-in variables invoked through standard calls
  • User-defined data structures used within the shader programs
  • Message passing architecture
    • To pass arbitrary data from client code (C/C++) to Shader
    • To pass data between component shader programs
  • Shader component programs
    • Most commonly vertex and/or pixel shaders
      • Invoked at specific locations in the pipeline on many data elements in
        parallel (e.g. pixel, fragment, geometry)
    • “Techniques” (Cg, HLSL) – not covered in this lecture
      • Describe grouping of vertex and pixel shader
      • Describe ordering of same

Shader Component Programs

• Vertex shaders
  • Executed once per vertex in a scene.
  • Transforms 3D position in space to 2D coordinate on screen
  • Can manipulate position, color, texture coordinates
  • Cannot add new vertices

• Fragment shaders (pixel shaders)
  • Calculates the color of individual pixels
  • Used for lighting, texturing, bump mapping, etc.
  • Executed once per pixel per geometric primitive

• Geometry shaders
  • Can add/remove vertices from a mesh
  • Can procedurally generate geometry, or add detail to shapes
Summary from last lecture (1)

Graphics Processing Unit (GPU):
- Replaces traditional "fixed function" graphics pipeline
- Massively parallel, special purpose SIMD processor
- Executes on thousands of threads concurrently

Shader programs (GLSL):
- Vertex shaders
  - Executes once per vertex
  - Input: vertex & associated data from C code (e.g. glVertex*)
  - Minimal output: updated vertex position, color
- Fragment shaders
  - Executes once per pixel or sub-pixel region
  - Input: transformed data from vertex shader
  - Typically fragment shader invocations >> vertex shader invocations
  - Typically by order of magnitude
  - But not always! (Data sensitive)
- Output: color value to framebuffer
- Each has valid main() entry point
- No pointers on direct memory access (with major exception discussed shortly!)

Summary from last lecture (2)

Understanding GLSL Shader program structure
- Variable qualifiers:
  - Determine variable scope & access capability of global variables
  - Essential for communicating between program components
    - C-code to vertex shader
      - Vertex shader to fragment shader
    - 3 qualifier types:
      - Uniform
        - Visible to both vertex & fragment shaders
        - Do not change within a mesh declaration, e.g. set prior to glUseProgram() & bind() block
        - Can be written only by calling program (C code), cannot be changed by shader programs
      - Attribute
        - Visible to vertex shaders, but not fragment shaders
        - Typically set once per vertex, e.g. make a call to change attrib. before glVertexAttribPointer() call
        - Can be written only by calling program (C code), cannot be changed by shader programs
      - Varying
        - Not visible to C code program
        - Typically set once per vertex shader invocation
        - Can only be written in vertex shader and read by fragment shader
        - Values interpolated automatically for each fragment during rasterization.

Summary from last lecture (3)

Understanding GLSL Shader program structure (cont)
- Data access & communication:
  - Shader program variables stored in a lookup table
  - Can change values in client (C code) program
- 2-step process:
  - Ask GL for variable by name:
    - glGetUniformLocation & glUseProgram
    - Parameters: shader program handle, "var name"
  - Returns numeric ID (handle) to variable in table
- Set the variable state:
  - glGetUniformLocation, glUniform
  - Function has type qualified prototype (e.g. 3,5,etc.)
- Parameters: handle to shader variable, type, correct value
  - Applies only to Uniform, Attribute qualified

Summary from last lecture (4)

Understanding GLSL Shader program structure (cont)
- Shader program initialization:
  - Obtain an empty program shell from GL: glCreateProgram()
  - Shader program encapsulates vertex shaders, fragment shader
  - Must add vertex, fragment shaders to it (typically 2 each)
  - For each vertex, fragment shader:
    - Read in code (and source file) under a function called ADDITIVE
      - Get a fresh handle: glGetUniformv()
      - GL_VERTEX_SHADER
      - GL_FRAGMENT_SHADER
  - Give the shader source code: glShaderSource()
  - Compile the source code: glCompileShader()
  - Attach to the program shell: glLinkProgram()
  - Link the program: glLinkProgram()
  - Tell GL to actually use it (glUseProgram())
    - At the right moment in your code:
      - Can turn on/off selectively
      - One C program can use many shaders.