Shader Combining with NVLINK & NVASM

Chris Maughan

Problem Statement

- Typically require many shading effects on one set of polygons
  - Custom Lighting
  - Custom Transform
  - Pixel shader setup (basis vectors, etc.)
- Combining all relevant effects in one shader is tricky
  - Combinatoric nightmare
  - Limited resources (128 instructions, 12 Registers, etc.)

Solution

- NVASM + NVLINK
  - NVASM assembles shader files into ‘fragments’
  - NVLINK combines fragments into shaders
- Result is a combined shader that achieves result and runs on target hardware
- Design goals
  - Capable of generating shaders in-game (fast!)
  - If at all possible, can make shader fit into hardware limits (registers/instructions/constants)
  - Easy to author fragments

NVASM

- NVASM creates .nvo object files from .nvf files
  - Use ‘–f’ switch to enable fragment generation
- Typical fragment:

```
#beginfragment world_transform
dp4 r_worldpos.x, v_position, c_world0
dp3 r_worldpos.y, v_position, c_world1
dp4 r_worldPos.z, v_position, c_world2
#endfragment
```

NVASM code structure

- Fragment files can make use of ++, -- operators on constants

```
dp3 r_lightintensity, r_normal, c_thislight++
add r_totallight, r_lightintensity, r_totallight
```

- This enables including the same fragment multiple times, and auto-incrementing the variable that is used (useful for items like lights)
- New constants created this way have the same ID, but a different offset, so you can ask the linker for each instance

NVASM code structure (2)

- Fragments are named in file
  - (#beginfragment,#endfragment)
- Use symbol names for registers that are not defined (e.g. c_world, v_pos)
  - Assigned to real registers during link
  - Can use standard register names for fixed locations – linker will not re-assign fixed registers
    - Useful if you have, say, a fixed constant-map
    - But this reduces the linker's ability to efficiently combine fragments

Offline process - NVASM (3)

- NVASM outputs .nvo files
  - Contain shader fragments
  - Contain symbol table
  - These are loaded by NVLINK
- NVASM is an off-line process
  - Parsing/Macro processing is done off-line
  - Shader syntax validation is done off-line
- Fragment files can not contain standard vertex/pixel shaders, only fragments

NVLINK

- Supplied as .dll for in-game shader generation
- Linking is a two-step process
  - Step 1 – Give the linker a list of all fragment files (1 file may contain several fragments). This is done at game start – not during the scene!
  - Step 2 – Ask the linker to generate a shader, based on a list of fragment ID’s (retrieved from Step 1)

NVLINK – Pre-Process

- Supply a list of .nvf files
  - .nvf files contain fragments that are ‘logically’ dependant:
    - linker object assumes all fragments passed to it contain symbols in the same ‘namespace’
      - c_lightdirection in file ‘lights.nvf’ and c_lightdirection in file ‘characterlight.nvf’ are the same symbol…
    - Create another linker object if you have ‘sets’ of shaders – ‘space station shaders’, ‘underground shaders’, etc.
- Pre-built sets of shaders enable fastest run-time link performance

NVLINK – Pre-Process (continued)

- Request list of fragment ID's:
    - GetFragmentID(char* pName)
  - Linker returns fragment ID
  - Arrays of ID's are passed to linker to request a shader create, eg:
    - ID 0 = xform_eye_space
    - ID 1 = xform_normal
    - ID 2 = light_eye_space_directional
    - ID 3 = light_eye_space_point
    - [0,1,2] = Eye space directional lighting
    - [0,1,3] = Eye space point lighting

NVLINK – Pre-Process (continued)

- **Request list of constant ID's:**
    - **GetConstantID(char* pName)**
  - **Linker returns constant ID**
  - **ID is used after creating a shader to find out where the constant was allocated (i.e. the constant location required)**
- **Request list of vertex ID's:**
    - **GetVertexID(char* pName)**
  - **Linker returns vertex ID**
  - **ID is used after creating a shader to find out where the vertex was allocated (i.e. the stream location required)**

NVLINK – Link Phase

- Generate Shader
  - pShader = CreateBinaryShader(&hShader[0], &pBuffer);
    - Returned buffer is an NVLinkBuffer – similar semantics to D3DXBuffer
    - Pass returned buffer to CreateVertexShader
      - Shader will be validated by runtime - *this may well take longer to run than the link phase!*
    - Call ->Release() on NVLinkBuffer
- Can call GetShaderSource() to get the sources for the last generated shader – useful for debug, but not fast!

NVLINK – Post-Link

- **Call GetConstantSlot(ID, Offset, DWORD* pSlot) / GetVertexSlot(ID, DWORD* pSlot)**
  - **Stores slot in constant/vertex memory in ‘pSlot’ E.g. *pSlot = 3 means constant with this ID goes in slot 3**
  - **Some generated shaders may not require ‘v_diffuse’ (for example), so you can generate vertex data that does not send it**
  - **Letting the linker generate vertex and constant slots for you gives more flexibility for optimizations, but is more complex to code your app**

NVLINK – What it does (1)

- Pre-Process
  - Resolves global symbol table for all fragments
    - Hence fragments are semantically grouped
  - Prepares internal fix-up lists for symbols in fragments
    - Enables fast location of symbol relocations required
  - Generates ‘scope’ for parameters in fragment instructions
    - Enables variable re-use at link stage – at the per-register component level (e.g. r1.x)
  - Several other pre-process steps to ensure link phase is fast as possible

NVLINK – What it does (2)

- Link-Phase
  - Given set of fragment ID’s, splices together fragment binaries
  - Walks the shader assigning registers and retiring unused registers for re-use
    - Resulting code can look very different to what you supplied, depending on how many symbols you used
    - More symbols give more flexibility in linking, but slightly slower link performance (but target is to handle many symbols very quickly)
  - Generates lists of registers assignments for constant and vertex streams

NVLINK – What it does (3)

- Link-Phase continued…
  - *May* remove obvious redundant code

    mov r0, r1 << redundant

    Mad r4, r2, r3, r0 << replace r0 with r1
- Why not do this?
  - Because it will replicate work done in the driver
- Why do this?
  - Because it may be necessary to fit the code in the available instruction memory…

NVLINK Performance

- Architected to be fast at the link phase, slow at the preparation phase
- Performance measurements will be available
- Aim is to be fast enough to use during game-scene
    - Limiting factor may be performance of CreateVertexShader(…)
        - Causes work to be done in the driver preparing the shader for the chip
        - Also causes work to be done in the runtime

NVLINK Demo Effect

- Built into NVEffectsBrowser
- Demo
  - T&L Pipeline demo, with specular/diffuse, point/directional, eye space/object space, blinn bump setup, etc.
    - Useful as a demo of how to build a fixed-function equivalent shader
    - Employ advanced mesh builder to handle dynamic allocation of constant/vertex stream data
    - Benchmark option available to test create time of shader

Questions, comments, feedback?

- Version 1.0 shipping (www.nvidia.com/developer.nsf)
- Feedback welcome
- Chris Maughan, cmaughan@nvidia.com
- Questions?