Programmable shaders

Computer Graphics
COMP 770 (236)
Spring 2007

Instructor: Brandon Lloyd
From last time…

- Evolution of graphics pipeline
- High level languages for GPU programming
- GPU hardware architecture
Today’s topics

- Flexible shading
  - shading networks
  - shading languages

- GLSL
  - language features
  - interfacing shaders with the application
Shading networks

- Hook up parameters and operations freely

- concatenate
  - before
  - during
  - after

- bump mappin
  - offset
  - color

- plastic
  - diffuse
  - specular

- marble
  - frequency
  - amplitude

- texture

- constant
  - constant
  - constant
Shading languages

- Well defined interface with the rendering program
  - rendering program provides all the required inputs for the shader
  - shader produces specified outputs
- Inputs can be manipulated with general programming constructs
- Built-in functions and constructs enable shading operations to be expressed succinctly
Shader networks v. shader programs

- Shader networks are easier for non-experts to use
- Shader programs are more flexible
- Shader programs can become components of shader networks
  - often done in production. Programmers create components for artists to use
  - allows for greater reuse
- Abstract syntax tree built by compiler can be seen as a shader network
  - but not directly useful
RenderMan

- Created by Pixar in 1988
- Has become an industry standard
- Multiple shader types
  - light source, surface, volume, displacement
- Uniform v. varying variables
- Designed for SIMD execution

```
surface
dent( float Ks=.9, Kd=.5, Ke=.1, roughness=.25, dent=.4 )
|
  float turbulence;
  point N, V;
  float i, freq;
  
  /* Transform to solid texture coordinate system */
  V = transform("shader", P);

  /* Sum 6 octaves of noise to form turbulence */
  turbulence = 0; freq = 1.0;
  for( i=0; i<6; i++ ) {
    turbulence += 1/freq * abs( 0.5 - noise( i*freq*V ) );
    freq *= 2;
  }

  /* Sharpen turbulence */
  turbulence *= turbulence * turbulence;
  turbulence -= dent;

  /* Displace surface and compute normal */
  P = turbulence * normalize(N);
  N = faceforward( normalize(calculateNormal(P)), 1 );
  V = normalize(-T);

  /* Perform shading calculation */
  Gd = 1 - smoothstep( 0.03, 0.05, turbulence );
  Cl = Gd * Cs = (Ke*ambient()) + Ke*specular(Kf,V,roughness))
```
GPU shading languages

- **CG**
  - NVIDIA’s shading language
  - works on OpenGL and DirectX
  - Uses hardware profiles that may limit language constructs

- **HLSL**
  - Shading language used in DirectX
  - very similar to CG

- **GLSL**
  - OpenGL’s built in shading language

- **Sh**
  - A C++ library rather than a language
  - Can cross-compile to the GPU
GLSL data types

- **Scalar data types**
  - float, bool, int

- **Vector data types**
  - vec{2,3,4} – float vector
  - bvec{2,3,4} – bool vector
  - ivec{2,3,4} – int vector
  - can be accessed with [] operator
    
    ```
    float4 color = vec4(1,2,3,4);
    float red = color[0];
    ```
  - can also be accessed with (r,b,g,a), (x,y,z,w), or (s,t,p,q) members
    
    ```
    red = color.r  or  red = color.x  or  red = color.s
    ```
  - “swizzling”
    
    ```
    float4 v1 = color.abgr
    float4 v2 = color.rraa
    float2 v3 = color.zw
    ```
GLSL data types

- **Matrix data types**
  - mat2, mat3, mat4
  - there are others as well

- **Sampler data types – for texture access**
  - sampler1D, sampler2D, sampler3D, samplerCube
  - sampler1DShadow, sampler2DShadow – for shadow maps

- **Arrays**
  - similar to C
  - must be fixed length

- **Structs**
  - same syntax as C
GLSL variable qualifiers

- **const**
  - compile time constant

- **uniform**
  - variables that have the same value for every point

- **attribute**
  - variables that are specified per vertex

- **varying**
  - variables that can change for every fragment
  - interpolated from vertex attributes
GLSL control flow

- Has the standard C constructs:
  - if ( bool expression ) { ... } else { ... }
  - for ( initializer; bool expression; loop expression ) { ... }
  - while( bool expression ) { ... }
  - do { ... } while ( bool expression )
  - continue
  - break

- GLSL adds “discard” which halts execution of the current fragment

- Hardware-dependent limitations
  - loop may be required to have a fixed number of iterations
  - both the if clause and the else clause may be executed
GLSL functions

- Syntax like C functions
- Function arguments may have qualifiers
  - in, out, inout
- Each type of program (e.g. vertex, fragment) must have a main() function
- There are many built-in functions for computing various mathematical operations

Language spec and quick reference guide:
  - http://www.opengl.org/documentation/gls/
Demos

wavy teapot

checker teapot
One or more shaders are attached to a program

- Can have multiple shaders of one type, but at least one of them must have main()

A single shader can be used in multiple programs.
Passing parameters

- **Attribute variables**
  - `glProgramiv()` – use to find the number of active attributes
  - `glGetActiveAttrib()` – returns info about parameters used
  - `glGetAttribLocation()` – determine which attribute a variable is assigned to
  - `glBindAttribLocation()` – manually specify which location to use for a particular variable
  - `glVertexAttrib*()` – specify generic vertex attribute

- **Uniform variables**
  - `glProgramiv()` – use to find the number of active uniforms
  - `glGetActiveUniform()` – returns info about parameters used
  - `glUniform{1,2,3,4}{ifv}()` – pass parameter to shader
  - Samplers are passed with `glUniform1i()`
GLSL wrapper

- Martin Christen has a nice GLSL wrapper than can hide a lot of this complexity
  - libglsl
  - http://www.clockworkcoders.com/oglsl/
Next time

- Visibility computations