# Introduction to GLSL

EECS 487

January 29, 2006

# project 2

- start now
- note: I'm adding better documentation to support code today...
- read phorum
- for GLSL:
  - use Autolab or Cooley
  - or your own computer if gfx card is good
  - other labs?

# homework 1

- Will be posted this afternoon

# Next...

- Finish slides from last time,
- then introduce GLSL programming

# flow control in jot

GL_VIEW renders scene: geom/gl_view.H

1. clear buffer
2. initialize OGL state (default values)
3. setup lights (see code example in p2.C)
   (send light coords to OGL)
4. draw objects

# drawing objects

Loop over list of GELs (disp/gel.H)

GEL = “geometric element”

virtual method: GEL::draw()

generic scene object, includes:

- 2D objects like text in window corner
- 3D objects that contain meshes

subclass GEOM contains a mesh

# drawing a GEOM

For each GEOM:

send material properties to OGL

send transform to OGL

draw the mesh

BMESH class represents a mesh (mesh/bmesh.H)

Essentially: vertices, edges, faces

# drawing a mesh

BMESH may be divided into patches,
   each patch rendered separately

Common case: entire mesh is 1 patch

```
BMESH::draw(){
  for each patch
     draw the patch
}
```

Patch class represents a patch
   (mesh/patch.H)

# drawing a patch

```
Patch::draw() {
    check name of current rendering style
    find GTexture with matching name
    tell GTexture to draw
}
```

In modern terms, GTexture is a “shader”

AKA “procedural texture”

“Generalized texture” ... “groovy texture”? something like that...

# Why keep a list of GTextures?

- Patch keeps a list of GTextures, but only uses 1 at any given time
- Reason: GTextures may contain data
  - When switching styles, don't want to destroy data from previous style
  - This way, switching styles is lightweight

# drawing a GTexture

Details vary per GTexture

Common case:

- setup OGL state, e.g.:
    - enable or disable lighting
    - enable or disable alpha blending, etc.
- draw triangle strips
    - use StripCB (mesh/stripcb.H)

# StripCB

- lets you customize drawing of triangle strips
- while iterating over a triangle strip:
  - almost always call glVertex()
  - sometimes call glNormal()
  - sometimes call glTexCoord()
  - sometimes call glColor()
- use StripCB subclass to make whatever combination of calls is needed

# accessing material properties

- Patch is a subclass of APPEAR (disp/appear.H), which stores all the material properties.

- you’ll need that info in your software shader

# Next: GLSL

# OpenGL Pipeline

1. vertex processing
   - transformations: 3D → 2D
   - lighting
2. clipping, primitive assembly
3. fragment processing
   - rasterize primitives
   - interpolate colors, texture coordinates, etc.
4. fragment test, etc.
   - depth, alpha
   - alpha blending

# Programmable parts

- vertex processing
  - transformations: 3D → 2D
  - lighting
- clipping, primitive assembly
- fragment processing
  - rasterize primitives
  - interpolate colors, texture coordinates, etc.
- fragment test, etc.
  - depth, alpha
  - alpha blending

# Basic idea

- Replace vertex or fragment computations with application-provided *programs*
  - also called *shaders*
- Written in high-level language: GLSL
- Graphics driver compiles and links program at run-time
- Application activates the program to replace fixed-functionality OpenGL pipeline

# 2 issues

1. How to write shaders
2. How to activate shaders in OpenGL

our focus: #1

jot handles #2

- nothing deep; read the manual

# GLSL: C Basis

- Based on C, with some C++ features
- Graphics-friendly data types: `vec2, vec3, vec4, mat2, mat3, mat4, void, bool, float, int,` ...
- structs, 1D arrays, functions, iteration, if/else

# Code snippet

```
void main() {
   const float f = 3.0;
   vec3 u(1.0), v(0.0, 1.0, 0.0);
   for (int i=0; i<10; i++)
      v = f * u + v;
   …
}
```

# General purpose?

- Seems like general purpose computing.
  - Anything missing?

# Missing features

- No pointers or dynamically allocated memory
- No strings, characters
- No double, byte, short, long, unsigned…
- No file I/O
- No printf()
- Focus is numerical computation

# Other differences

- No automatic type conversion

```
float f = 1; // WRONG
 float f = 1.0; // much better
```

- Simplifies things
- Instead of casting, use constructors:

```
vec3 v3 = vec3(0.5, 1.0, 0.5);
vec4 v4 = vec4(v3, 1.0);
vec2 v2 = vec2(v4);
float f = float(1);
```

# Other differences

- 3 kinds of function parameters:
  - **`in`** (assumed)
  - **`out`**
  - **`inout`**
- no pointers or references

# Graphics-friendly functions

- `sin, cos, tan, asin, acos, atan, ...`
- `pow, exp2, log2, sqrt, …`
- `abs, floor, ceil, mod, min, max, clamp…`
- `mix, step, smoothstep`
- `length, distance, dot, cross, normalize`
- `reflect (!)`
- more…

# Type qualifiers

Variables passed to shaders from the application:

`uniform:`

- value is constant over primitive (e.g. light direction)

`attribute:`

- value varies per-vertex (e.g. vertex normal)
- built-in (e.g. `gl_Vertex`) or application-specific

`varying:`

- output from a vertex shader
- input to a fragment shader
- (interpolated per-fragment)

# Examples: per-pixel lighting

Switch to browser to examine vertex and fragment shaders provided in project 2 support code:

- lighting.vp
- lighting.fp

# Online resources

http://developer.3dlabs.com/openGL2/

http://www.lighthouse3d.com/opengl/glsl/

http://www.opengl.org/documentation/glsl/