Computer Graphics Programming II

- Agenda:
  - Course road-map
  - Introduce OpenGL Shading Language (GLSL)
    - Overview of programmable GPUs
    - GLSL syntax
    - Using GLSL shaders
  - Phong shading with GLSL

What should you already know?

- All of the prerequisites from VGP351:
  - C++ and object oriented programming
    - For most assignments you will need to implement classes that conform to a very specific interface.
  - Graphics terminology and concepts
    - Polygon, pixel, texture, infinite light, point light, spot light, etc.
  - Some knowledge of linear algebra / vector math
    - Dot product, cross product, vector addition, subtraction, etc.
  - Some calculus will help with the readings

What should you already know?

- Drawing with OpenGL's fixed-function pipeline.
  - Setting transformations
  - Submitting vertex data
  - Enabling and controlling lights
  - Loading and configuring textures
  - Enabling and controlling texture environment
- Using OpenGL extensions

What will you learn?

- OpenGL Shading Language

What will you learn?

- OpenGL Shading Language
  - How to write shaders.

What will you learn?

- OpenGL Shading Language
  - How to write shaders.
  - Load, compile, link, and use shaders.

What will you learn?

- OpenGL Shading Language
  - How to write shaders.
  - Load, compile, link, and use shaders.
- Advanced lighting models

What will you learn?

- OpenGL Shading Language
  - How to write shaders.
  - Load, compile, link, and use shaders.
- Advanced lighting models
  - BRDFs for realistic rendering of real materials

What will you learn?

- OpenGL Shading Language
  - How to write shaders.
  - Load, compile, link, and use shaders.
- Advanced lighting models
  - BRDFs for realistic rendering of real materials
  - Rendering fur and hair

What will you learn?

- OpenGL Shading Language
  - How to write shaders.
  - Load, compile, link, and use shaders.
- Advanced lighting models
  - BRDFs for realistic rendering of real materials
  - Rendering fur and hair
  - “Toon” and other non-photorealistic rendering

What will you learn?

- OpenGL Shading Language
  - How to write shaders.
  - Load, compile, link, and use shaders.
- Advanced lighting models
  - BRDFs for realistic rendering of real materials
  - Rendering fur and hair
  - “Toon” and other non-photorealistic rendering
  - Procedural textures

How will you be graded?

- Tests and quizzes:
  - Bi-weekly quizzes worth 5 points each
  - A final exam worth 50 points
- Programming assignments:
  - Seven **weekly** programming assignments worth 10 points each
    - Each of assignment builds on the previous assignment
  - One three-week term project worth 50 points
- One in-class presentation worth 10 points

How will programs be graded?

- First and foremost, does the program produce the correct output?
- Are appropriate algorithms and data-structures used?
- Is the code readable and clear?

How will the presentation be graded?

- Read one of the papers during the term
  - You actually need to read *all* of them
- Present a summary of the paper to the class
  - What is the problem being solved?
  - How does the paper's author solve that problem?
  - What is novel about the author's solution?
  - What questions do *you* still have about the paper?

Per-fragment Lighting without GLSL

- Recap from last term...
  - Transform vertices, normals, and tangents *by hand*
  - Use transformed data to calculate *H* and *L* vectors *by hand*
  - Store *H* and *L* vectors in texture coordinates and / or colors
  - Configure texture environment to perform DOT3 on the bump map and *H* (specular) or *L* (diffuse).

Per-fragment Lighting without GLSL

- What's wrong with this technique?

Per-fragment Lighting without GLSL

- What's wrong with this technique?
  - Slow!
    - Lots of work to do on the CPU
    - New data per-frame → uploads and pipeline stalls
  - Difficult to implement
    - How many actually completed this last term? :)
  - Inflexible
    - Difficult to implement “shininess” exponents
    - Requires multiple passes for even simple effects

Root Causes

- Duplicate work that OpenGL already does
  - Re-transformation of vertex data
- Don't have access to the data that we really want in the texture combiners
  - Transformed light position
  - Transformed and interpolated normal

Programmable GPUs Solve This

- Vertex stage is programmable
  - Perform arbitrary calculations on per-vertex inputs
  - Pass arbitrary data to the fragment pipeline
  - Must *also* perform the “usual” vertex transformations
- Fragment stage is programmable
  - Perform arbitrary calculations on vertex stage outputs
  - Must generate output color
  - Can also modify fragment's Z value

Dependent Texturing

- Arbitrary values can be used to sample textures
  - Interpolated outputs of vertex stage
    - Just like fixed-function texture coordinates
  - Coordinates calculated by fragment shader
  - Value read from another texture
    - Use a displacement map to calculate an offset to an existing texture coordinate to read from another texture

What is GLSL?

- High-level, C-like shading language
  - Originally developed at 3dlabs
  - Part of core OpenGL in 2.0 (September 2004)
- Graphics oriented additions:
  - 2-, 3-, and 4-element vectors
  - 2x2, 3x3, and 4x4 matrices
    - OpenGL 2.1 adds non-square matrices
  - Special type qualifiers for shader inputs and outputs
  - Numerous built-in functions

Vertex Shader

- Programmable shaders replace the following:
  - Vertex transformation
  - Normal transformation, re-normalization, etc.
  - Lighting calculations
  - Texgen
  - Texture coordinate transformation

Vertex Shader (cont.)

- Programmable shaders do *not* replace the following:
  - Perspective calculations
  - Clipping
  - Backface culling
  - Primitive assembly
  - Polygon offset

Fragment Shader

- Programmable shaders replace the following:
  - All texture operations
  - Fog application
  - Application of primary and secondary colors
  - Other bits that we didn't use in VGP351.

Fragment Shader

- Programmable shaders do not replace the following:
  - Shading model (flat vs. smooth)
  - Alpha, depth, and stencil test
  - Alpha blending
  - Other bits that we didn't use in VGP351

Vector and Matrix Types

- 2-, 3-, and 4-element vectors of various basic types:
  - `bool` → `bvec2`, `bvec3`, `bvec4`
  - `int` → `ivec2`, `ivec3`, `ivec4`
  - `float` → `vec2`, `vec3`, `vec4`
- 2x2, 3x3, and 4x4 `float` matrices
  - `mat2`, `mat3`, `mat4`

Type Qualifiers

- Three special type qualifiers in GLSL
  - `uniform` – Shader inputs that are constant across a primitive group (begin / end pair).
    - Like the parameters specified via `glLightfv`, `glFogfv`, etc.
  - `attribute` – Vertex shader inputs specified per-vertex.
    - Built-in values like `glColor`, `glNormal`, etc
    - User-defined values
  - `varying` – Vertex outputs (fragment inputs) that are interpolated across primitives

Basic Vertex Shader

```
varying vec3 normal;

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

Basic Fragment Shader

```
varying vec3 normal;

void main(void)
{
    float dotProd = max(
        dot(gl_LightSource[0].position,
            normalize(normal)), 0.0);
    gl_FragColor =
      (gl_FrontMaterial.diffuse * dotProd)
      + (gl_FrontMaterial.specular
     * pow(dotProd, gl_FrontMaterial.shininess);
}
```

References

http://www.mew.cx/glsl_quickref.pdf

Break

Using Shaders – Overview

- There are a lot of steps, but it's not too scary.
  1. Create shader objects.
  2. Associate source code with shared objects.
  3. Compile objects.
  4. Attach objects to a program.
  5. Link program.
  6. Use the linked program!
- There is a *bit* more to it than this.

Create Shader Objects

- Create shader objects using `glCreateShader`

```
GLuint glCreateShader(GLenum type);
```

  - `type` is either `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`.
  - Unlike textures and buffer objects, this is the **only** way to create a shader.
- Create program object using `glCreateProgram`

```
GLuint glCreateProgram(void);
```

Set Shader Program Code

- Specify the source text for the shader

```
void glShaderSource(GLuint shader,
    GLsize count, const GLchar **code,
    const GLuint *length);
```

  - `shader` – Handle of the shader object whose source code is to be replaced
  - `count` – Number of elements in the `code` and `length` arrays
  - `code` – Array of pointers to strings containing the source code of the shader
  - `length` – Specifies an array of string lengths

Compile Shaders

- After specifying the program code, compile the shader:

```
GLvoid glCompileShader(GLuint shader);
```

  - Check for compile success with `glGetError`.
  - If the compilation fails, check the log with `glGetInfoLog`
    - See the manual page for the details

Link Program

- Attach vertex and fragment shaders to a program with `glAttachShader`

```
void glAttachShader(GLuint program,
    GLuint shader);
```

- Once all shaders are attached, link the program

```
void glLinkProgram(GLuint program);
```

  - After linking, check the error status and, if necessary, the log.
- A program need not have both a vertex shader and fragment shader

Use Linked Program

- Select *and enable* a program with `glUseProgram`

```
void glUseProgram(GLuint program)
```

  - Different from textures which have a separate bind and enable!

Break

Phong Shading

- Interpolate normals between vertices
  - If polygons are large, we will probably need to re-normalize the interpolated values.
- Interpolate *H* vector between vertices
  - Again with the re-normalize step
- Perform $(N \cdot H)^n$ per-fragment.

Surface-Space

- From the point of view of the surface (i.e., in *surface-space*), what is the normal vector?

Surface-Space

- From the point of view of the surface (i.e., in *surface-space*), what is the normal vector?
  - Assuming the surface is flat, N = (0, 0, 1).

Surface-Space

- From the point of view of the surface (i.e., in *surface-space*), what is the normal vector?
  - Assuming the surface is flat, N = (0, 0, 1).
- If we know the world-space surface normal, N_{surf}, can we create a transformation that will map N_{surf} to (0, 0, 1)?

Surface-Space

- From the point of view of the surface (i.e., in *surface-space*), what is the normal vector?
  - Assuming the surface is flat, N = (0, 0, 1).
- If we know the world-space surface normal, N_{surf}, can we create a transformation that will map N_{surf} to (0, 0, 1)?
  - Not uniquely.
  - If we knew another vector in the plane, we could create this transformation.

Tangents

- Call this new vector the *tangent vector*, and note it T_{surf}.
  - Knowing N_{surf} and T_{surf} is enough the create an orthonormal basis.
  - This basis can transform any vector into surface-space.
  - Tangent vectors can be created automatically (tricky) or by hand (annoying).

Where does H come from?

- NO WORK DONE ON CPU!!!
- In vertex shader:
  - Calculate the surface-space transformation
  - Calculate *H* per-vertex
  - Transform the per-vertex *H* vector to surface space
  - Pass *H* to fragment shader as a `varying`
- In fragment shader:
  - Re-normalize interpolated *H*

Where does N come from?

- Three ways to get *N*:
  - If surface is flat: N is constant (0, 0, 1), store in a combiner constant color.
  - If surface is curved: store per-vertex normal in one of the interpolated colors.
  - Surface is bumpy: fetch *N* from a texture.
    - Texture is stored so that R, G, and B map to the X, Y, and Z of the normal in surface space.
    - These textures tend to look blue because the Z component is usually close to 1.0.

Creating TBN Basis In GLSL

```
varying vec3 light_dir;
attribute vec3 tangent;

void main(void)
{
    gl_Position = ftransform();

    vec3 t = gl_NormalMatrix * tangent;
    vec3 n = gl_NormalMatrix * gl_Normal;
    vec3 b = cross(n, t);

    vec3 vert_pos = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 light = gl_LightSource[0].position - vert_pos;
    vec3 l;

    l.x = dot(light, t);
    l.y = dot(light, b);
    l.z = dot(light, n);
    light_dir = normalize(l);
}
```

Next week...

- More GLSL
  - User defined uniforms
  - User defined attributes
- Render to texture
- Environment mapping
- Assignment #1 due

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.