Today’s class

- OpenGL Shading Language
Shading Languages

- Pixar’s RenderMan shading language is the first to look at shaders as mathematical expressions.
- Expressions involve variables, constants, and operations among them.
- Some of these are available from the graphics environment.
- Some common shading languages:
  - OpenGL Shading Language (GLSL)
  - Cg (C for Graphics)
OpenGL Shading Language

- Vertex shaders
- Fragment shaders
- Part of OpenGL 2.0
Vertex shaders

- Replaces the fixed-function operations performed by the vertex processor with operations defined by the shader
- Executed on each vertex as it passes down the graphics pipeline
- Must output the information the rasterizer needs to do its job
- Minimum information that needs to be output:
  - Vertex position
What happens to vertices?

- Vertex positions are defined in object (world) space.
- Transformed first by the model-view matrix into eye coordinates.
- Then transformed by the projection matrix into clip coordinates.
- Thus, vertex shaders must perform this set of transformations.
- Vertex program can access the OpenGL state.
Simple vertex shader

/* pass through vertex shader */
void main (void)
{
    gl_Position =
        gl_ProjectionMatrix *
        gl_ModelViewMatrix *
        gl_Vertex;
}

A second simple vertex shader

/* simple vertex shader */
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main (void)
{
    gl_Position =
      gl_ModelViewProjectionMatrix * 
      gl_Vertex;
    gl_FrontColor = red;
}
Fragment shaders

- Same syntax as vertex shaders
- Executed after the rasterizer
- Operate on each fragment rather than on each vertex
Simple fragment shader

/* simple fragment shader */
void main (void)
{
    gl_FragColor = glFrontColor;
}

GLSL data types

- **Vectors**
  - vec2, vec3, vec4 *(floating-point)*
  - ivec2, ivec3, ivec4 *(integer)*
  - bvec2, bvec3, bvec4 *(Boolean)*
  - Can be indexed by position (x,y,z,w), color (r,g,b,a), or texture coordinates (s,t,p,q)

- **Matrices**
  - mat2, mat3, mat4
  - Always square and floating point
GLSL qualifiers

- **const** – makes the variable unchangeable by the shader
- **attribute** – used by vertex shaders for variables that change at most once per vertex in the vertex shader
- **uniform** – used for variables that are set in the application program for an entire primitive (variables whose values are assigned outside the scope of a `glBegin` and a `glEnd`)
- **varying** – provide the mechanism for conveying data from a vertex shader to a fragment shader (defined on a per-vertex basis but are interpolated over the primitive by the fragment shader)
GLSL operators

- Operations are overloaded so you can do matrix and vector operations (such as multiplication using `*`)
- Swizzling allows selection of multiple components from a matrix or vector
  - Example: if \( \mathbf{a} \) is declared as a \( \text{vec}4 \), then \( \mathbf{a}.xy = \mathbf{a}.yx \); swaps the \( x \) and \( y \) values of \( \mathbf{a} \)
GLSL functions

- GLSL contains the usual math and trig functions, such as \( \sin, \cos, \tan, \sqrt{\text{sqrt}}, \ \text{abs}, \ \text{etc.} \)

- Additional functions to help with geometric calculations involving vectors, such as \( \text{length, distance, normalize, and reflect} \)
Linking shaders to OpenGL programs

- Need a program object that can hold the shaders:
  
  ```
  GLuint myProgObj;
  myProgObj = glCreateProgram();
  ```

- Need an object to hold the shader:
  
  ```
  GLuint myVertexObj;
  myVertexObj = 
  glCreateShader(GL_VERTEX_SHADER);
  ```
Linking shaders to OpenGL programs

- Need to tell OpenGL where the shader source is:
  ```c
  glShaderSource(myVertexObj, 1, vertexProg, NULL);
  ```
  where `vertexProg` contains the shader code

- Need to compile the shader:
  ```c
  glCompileShader(myVertexObj);
  ```

- Need to attach the shader to the program object:
  ```c
  glAttachObject(myProgObj, myVertexObj);
  ```
Linking shaders to OpenGL programs

- Need to identify which program object to use:
  
  ```
  glUseProgram(myProgObj);
  ```

  where `vertexProg` contains the shader code

- Need to link everything together:
  
  ```
  glLinkProgram(myProgObj);
  ```
Example program

- shader.c is an example program that demonstrates a simple OpenGL shader
Shader to move vertices sinusoidally

```
uniform float time;
    /* value provided by application program */

void main()
{
    float s;

    s = 1.0 + 0.5*sin(time);
    gl_Position = vec4(s, s, s, 1.0) *
        (gl_ModelViewProjectionMatrix * gl_Vertex);
}
```
Aligning variables with shaders

- Need to align variables used in shaders with variables used in application program so information can be passed between them.
- We’ll increment a time variable in our idle callback function that needs to be passed to the vertex shader just described.
Aligning variables with shaders

- Shader variables are indexed in the main program through tables that are set up during linking.
- Need to get the location of the variable in the table:

  ```
  Glint timeParam;
  timeParam = glGetUniformLocation(program, "time");
  ```
Aligning variables with shaders

Now you can send the value of the program variable (let’s say it’s `myTime`) to the shader:

```gl
glUniform1f (timeParam, myTime);
```
Example program

- Program movingShader.c demonstrates the passing of variables to shaders