Computer Graphics (CS 543)
Lecture 3b: Shader Setup & GLSL Introduction

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

OpenGL function format

function name

Number of arguments

`glUniform3f(x,y,z)`

belongs to GL library

`x,y,z` are `floats`

`glUniform3fv(p)`

Argument is array of values

`p` is a pointer to array

Lack of Object Orientation

- OpenGL is not object oriented
- Multiple versions for each command
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`

OpenGL Data Types

C++	OpenGL
Signed char	GLByte
Short	GLShort
Int	GLInt
Float	GLFloat
Double	GLDouble
Unsigned char	GLubyte
Unsigned short	GLushort
Unsigned int	GLuint

Example: Integer is 32-bits on 32-bit machine
but 64-bits on a 64-bit machine

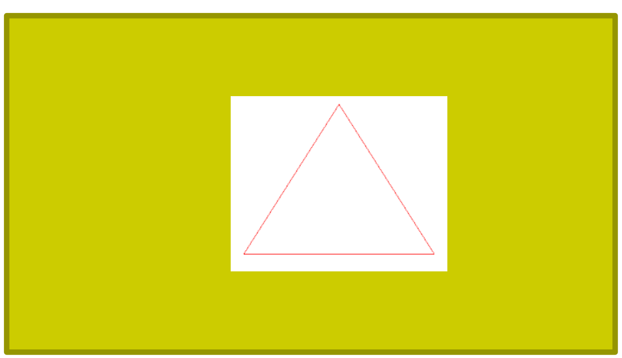
Good to define OpenGL data type: same number of bits on all machines

Recall: Single Buffering

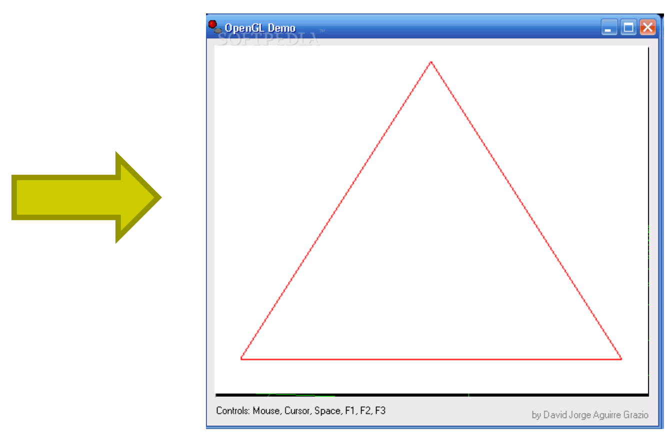
- If display mode set to single framebuffers
- Any drawing into framebuffer is seen by user. How?
  - `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);`
    - Single buffering with RGB colors
- Drawing may not be drawn to screen until call to `glFlush( )`

```
void mydisplay(void){
      glClear(GL_COLOR_BUFFER_BIT); // clear screen
      glDrawArrays(GL_POINTS, 0, N);
      glFlush( );
}
```

Drawing sent to screen



Single Frame buffer



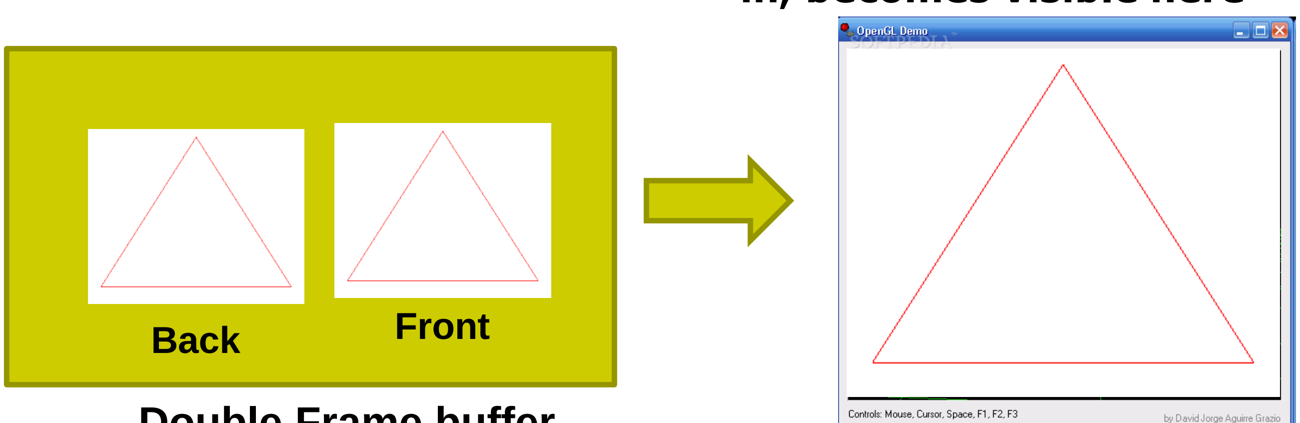
Double Buffering

- Set display mode to double buffering (create front and back framebuffers)
  - Double buffering is good for animations, avoids tearing artifacts
    - `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);`
      - Double buffering with RGB colors

- Front buffer displayed on screen, back buffers not displayed
- Drawing into back buffers (not displayed) until swapped in using **`glutSwapBuffers( )`**

```
void mydisplay(void){
        glClear(GL_COLOR_BUFFER_BIT); // clear screen
        glDrawArrays(GL_POINTS, 0, N);
        glutSwapBuffers( );
}
```

Back buffer drawing swapped in, becomes visible here



Double Frame buffer

Recall: OpenGL Skeleton

```
void main(int argc, char** argv){
   glutInit(&argc, argv);    // initialize toolkit
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(640, 480);
   glutInitWindowPosition(100, 150);
   glutCreateWindow("my first attempt");
   glewInit( );

   // … now register callback functions
   glutDisplayFunc(myDisplay);
   glutReshapeFunc(myReshape);
   glutMouseFunc(myMouse);
   glutKeyboardFunc(myKeyboard);

   glewInit( );
   generateGeometry( );
   initGPUBuffers( );
   void shaderSetup( );

   glutMainLoop( );
}
```

```
void shaderSetup( void )
{
   // Load shaders and use the resulting shader program
    program = InitShader( "vshader1.glsl", "fshader1.glsl" );
    glUseProgram( program );

    // Initialize vertex position attribute from vertex shader
    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0,
                                             BUFFER_OFFSET(0) );

    // sets white as color used to clear screen
    glClearColor( 1.0, 1.0, 1.0, 1.0 );
}
```

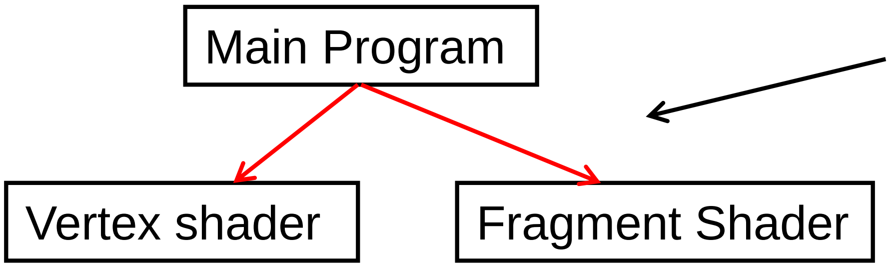
Recall: OpenGL Program: Shader Setup

- **`initShader( )`**: our homegrown shader initialization
  - Used in main program, connects and link vertex, fragment shaders
  - Shader sources read in, compiled and linked

```
Gluint = program;
```

```
GLuint program = InitShader( "vshader1.glsl", "fshader1.glsl" );
glUseProgram(program);
```

example.cpp



vshader1.glsl **fshader1.glsl**

What's inside **initShader?? Next!**

Coupling Shaders to Application (initShader function)

1. Create a program object
2. Read shaders
3. Add + Compile shaders
4. Link program (everything together)
5. Link variables in application with variables in shaders
   - Vertex attributes
   - Uniform variables

Step 1. Create Program Object

- Container for shaders
  - Can contain multiple shaders, other GLSL functions

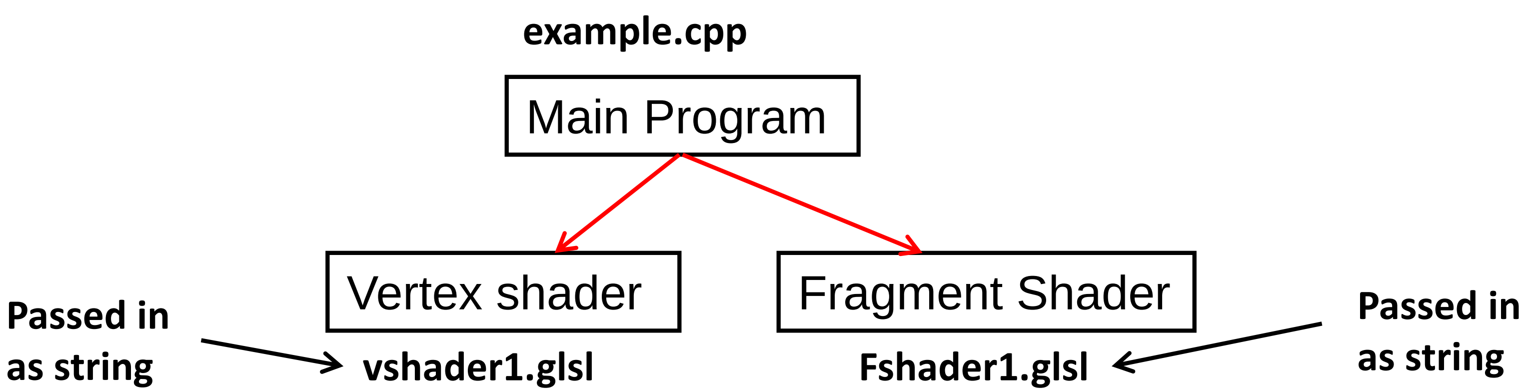
```
GLuint myProgObj;

myProgObj = glCreateProgram();
```

Create container called
Program Object

Main Program

Step 2: Read a Shader

- Shaders compiled and added to program object



- Shader file code passed in as null-terminated string using the function **`glShaderSource`**
- Shaders in files (vshader.glsl, fshader.glsl), write function **`readShaderSource`** to convert shader file to string

Shader Reader Code?

```
#include <stdio.h>

static char* readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");

    if ( fp == NULL ) { return NULL; }

    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);

    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);

    buf[size] = '\0';
    fclose(fp);

    return buf;
}
```

Shader file name (e.g. vshader.glsl) → **readShaderSource** → String of entire shader code

Step 3: Adding + Compiling Shaders

```
GLuint myVertexObj;
Gluint myFragmentObj;
```

Declare shader object (container for shader)

```
GLchar* vSource = readShaderSource("vshader1.glsl");
GLchar* fSource = readShaderSource("fshader1.glsl");
```

Read shader files, Convert code to string

```
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
myFragmentObj = glCreateShader(GL_FRAGMENT_SHADER);
```

Create empty Shader objects

example.cpp

Main Program

Vertex shader

vshader1.glsl

Fragment Shader

fshader1.glsl

Step 3: Adding + Compiling Shaders
Step 4: Link Program

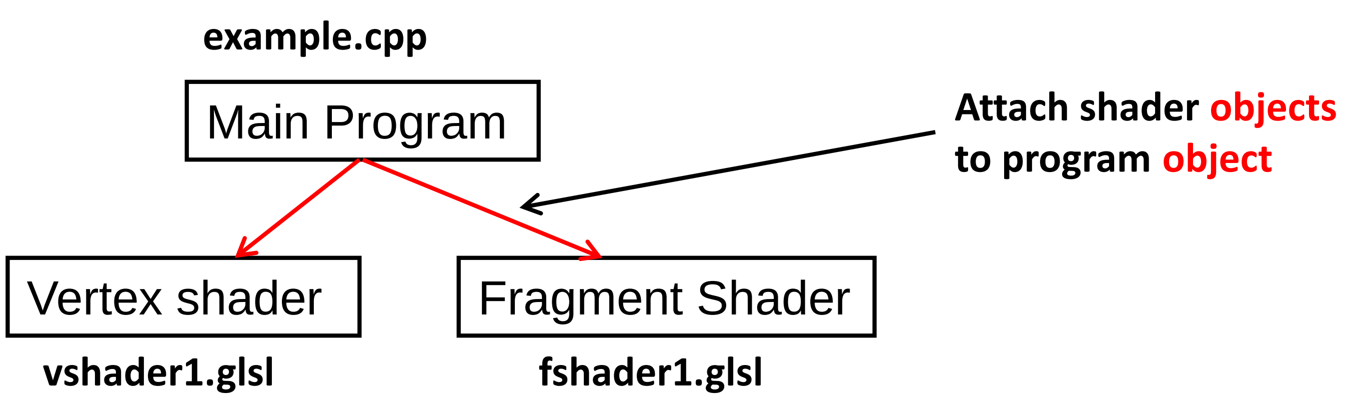
Read shader code strings into shader objects

```
glShaderSource(myVertexObj, 1, vSource, NULL);
glShaderSource(myFragmentObj, 1, fSource, NULL);

glCompileShader(myVertexObj);
glCompileShader(myFragmentObj);

glAttachShader(myProgObj, myVertexObj);
glAttachShader(myProgObj, myFragmentObj);

glLinkProgram(myProgObj);
```

Compile shader objects

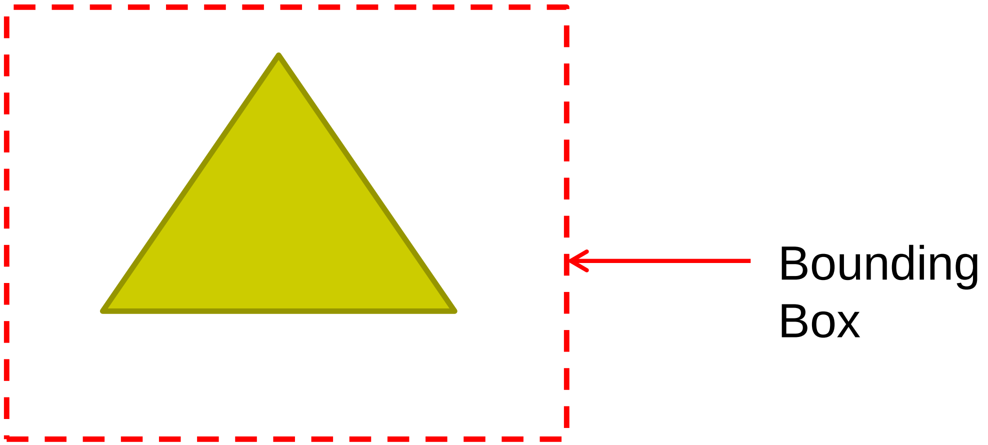
Attach shader objects to program object

Link Program



Uniform Variables

- Variables that are **constant** for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader
  - **Example:** bounding box of a primitive



Uniform variables

- Sometimes want to connect uniform variable in OpenGL application to uniform variable in shader
- Example?
  - Check “elapsed time” variable (etime) in OpenGL application
  - Use elapsed time variable (time) in shader for calculations

Uniform variables

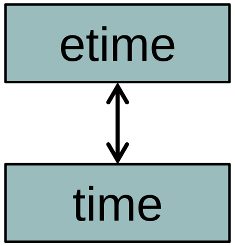
- First declare **etime** variable in OpenGL application, get time

```
 float etime;

etime = 0.001*glutGet(GLUT_ELAPSED_TIME);
```

Elapsed time since program started

- Use corresponding variable **time** in shader

```
 uniform float time;
attribute vec4 vPosition;

main(  ){
      vPosition.x  += (1+sin(time));
     gl_Position = vPosition;
}
```



- Need to connect **etime** in application and **time** in shader!!

Connecting **etime and time**

- Linker forms table of shader variables, each with an address
- In application, find address of shader **time** variable in linker table
- Tie **address of time** to application variable **etime**

```
Glint timeLoc;

timeLoc = glGetUniformLocation(program, “time”);
```

423	time

- Connect: **location** of shader variable **time** to **etime**!

```
glUniform1(timeLoc, etime);
```

Location of shader variable time

Application variable, etime

GL Shading Language (GLSL)

- GLSL: high level C-like language
- Main program (e.g. example1.cpp) program written in C/C++
- Vertex and Fragment shaders written in GLSL
- From OpenGL 3.1, application must use shaders

What does keyword out mean?

Example code of vertex shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
  gl_Position = vPosition;
  color_out = red;
}
```

gl_Position not declared
Built-in types (already declared, just use)

Passing values

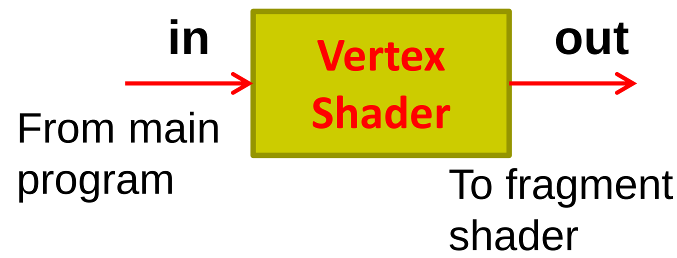
- Variable declared **out** in vertex shader can be declared as **in** in fragment shader and used
- Why? To pass result of vertex shader calculation to fragment shader

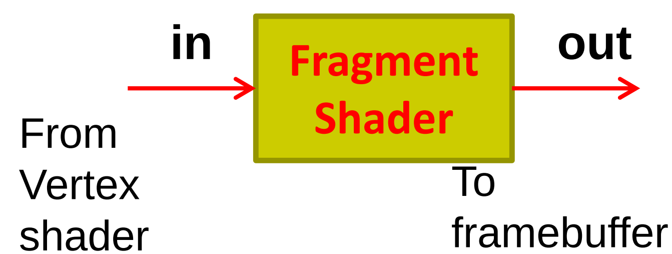
```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
  gl_Position = vPosition;
  color_out = red;
}
```

Vertex shader

```
in vec3 color_out;

void main(void){
  // can use color_out here.
}
```

Fragment shader





Data Types

- **C types:** `int, float, bool`
- **GLSL types:**
  - `float vec2: e.g. (x,y)          // vector of 2 floats`
  - `float vec3: e.g. (x,y,z) or (R,G,B) // vector of 3 floats`
  - `float vec4: e.g. (x,y,z,w)      // vector of 4 floats`

```
Const float vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out float vec3 color_out;

void main(void){
  gl_Position = vPosition;
  color_out = red;
}
```

Vertex shader

C++ style constructors (initialize values)

- Also:
  - int (ivec2, ivec3, ivec4) and
  - boolean (bvec2, bvec3,bvec4)

Data Types

- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- Matrices and vectors are basic types
  - can be passed in and out from GLSL functions
- E.g

  mat3 func(mat3 a)
- **No pointers** in GLSL
- Can use C structs that are copied back from functions

Operators and Functions

- Standard C functions
  - **Trigonometric:** cos, sin, tan, etc
  - **Arithmetic:** log, min, max, abs, etc
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a;     // a column vector stored as a 1d array
d = a*b;    // a row vector stored as a 1d array
```

Swizzling and Selection

- **Selection:** Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - **`vec4 a;`**
  - **`a[2], a.b, a.z, a.p`** are the same
- **Swizzling** operator lets us manipulate components

```
a.yz = vec2(1.0, 2.0);
```

References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition, Chapter 2
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 2