EECS 487: Interactive Computer Graphics

Lecture 19:
- Integrating shaders with OpenGL program
- Vertex-array objects with shaders
- Miscellaneous shader related stuff

Integrating GLSL with OpenGL

Two main topics:

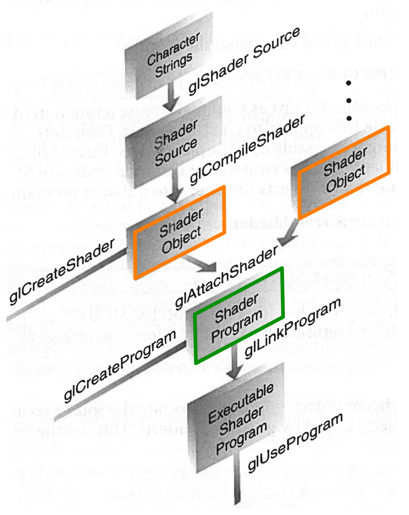
1. How do you get your shader codes into your OpenGL program?
   - in this course, all done for you in `shaders.cpp`
2. How do you pass data from your OpenGL program to your shaders?

Integrating GLSL with OpenGL

How do you get your shader codes into your OpenGL program?
- by runtime compilation to, and linking of, intermediate byte code

Topics:
- how to load shader source code into a running OpenGL program
- how to compile the source into byte code
- how to link multiple shaders into a shader program
  - there can be more than one shader programs per OpenGL program
- how to run the shader program

Integrating GLSL with OpenGL



First we need to create containers in your OpenGL program for the shader sources and shader programs

There are two types of "containers"
- shader object
- program object

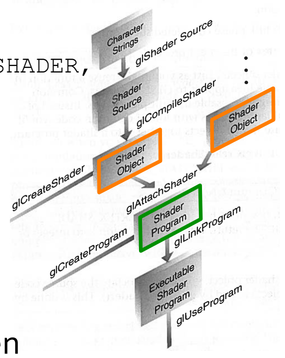
Onto the shader object:
- load source code from file
- compile source code into byte code

Onto the program object:
- attach shader objects
- link shader objects into a shader program

Use program

Creating Containers

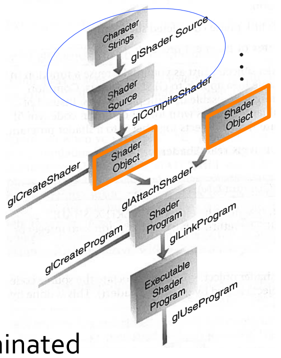
```
GLuint sd =
   GLuint glCreateShader(Glenum type);
// Allocates a shader container;
// type must be GL_VERTEX_SHADER,
// GL_FRAGMENT_SHADER, GL_GEOMETRY_SHADER,
// GL_TESS_CONTROL_SHADER,
// or GL_TESS_EVALUATION_SHADER

GLuint pd =
   GLuint glCreateProgram();
// Allocates a shader
// program container
```



If your OpenGL program crashes when calling `glCreateProgram()`, it most likely means your OpenGL set up doesn't support shaders

Load the Source

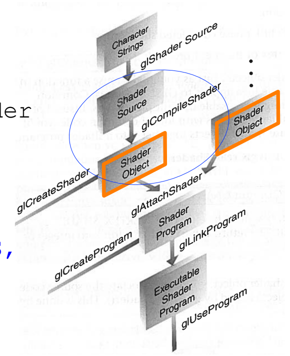
```
void glShaderSource(GLuint sd,
      GLsizei nstrings,
      const GLchar** strings,
      const GLint* lengths);
```



- `nstrings`: number of elements in the `strings` and `lengths` arrays
- if `lengths==NULL`, i.e., lengths not specified, `*strings` must be null-terminated
- need your own utility function to read shader source from file into `strings`

Compile the Source

```
void glCompileShader(GLuint sd);
// compiles the source code
// previously loaded for given shader

// check for success/failure using:
GLint status = 0;
glGetShaderiv(sd, GL_COMPILE_STATUS,
              &status);
if (status != GL_TRUE)
      // report compilation errors,
      // see next slide!
```

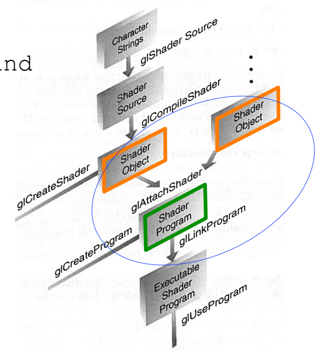


Report Compilation Errors

```
// Return the log associated with the last
// compilation of a shader
void glGetShaderInfoLog(GLuint sd,
                        Glsizei buf_size,
                        Glsizei* length,
                        char* info_log);

// e.g.,:
const GLsizei BUF_SIZE = 4096;
char info_log[BUF_SIZE] = {0};
GLsizei len = 0;
glGetShaderInfoLog(sd, BUF_SIZE, &len,
                   info_log);
cerr << "ShaderInfoLog: ” << endl
     << info_log << endl;
```

Attaching & Linking

```
void glAttachShader(GLuint pd,
                    GLuint sd);
// twice: once for vertex shader and
// once for fragment shader

void glLinkProgram(GLuint pd);
// After attaching all shaders,
// link program
// If no errors, program is
// ready for use

// check for errors:
GLint status = 0;
glGetProgramiv(pd, GL_LINK_STATUS, &status);
if (status != GL_TRUE)
   // see next slide
```



Report Linking Errors

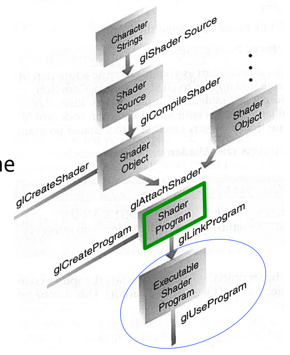
```
// Return the log associated with the last
// link attempt for a program:
void glGetProgramInfoLog(GLuint pd,
                         GLsizei buf_size,
                         GLsizei* length,
                         char* info_log);

// e.g.,:
const GLsizei BUF_SIZE = 4096;
char info_log[BUF_SIZE] = {0};
GLsizei len = 0;
glGetProgramInfoLog(pd, BUF_SIZE, &len,
                    info_log);
cerr << "ProgramInfoLog: " << endl
     << info_log << endl;
```

Running the Program

```
void glUseProgram(GLuint pd);
```

- switches on shader, bypasses fixed-function pipeline
- if `pid == 0`, shaders turned off

```
// e.g.,:
glColor3f(0.0,0.0,1.0);
// use shader:
glUseProgram(pd);
glBegin(GL_TRIANGLES);
   glVertex3i(1, 0, 0);
   glVertex3i(0, 1, 0);
   glVertex3i(0, 0, 1);
glEnd();
```



Fengo

- multiple shader programs may be created/loaded in a single OpenGL program and user can switch between them

Other Functions

Once you're done with a shader program, you can release the resources used:

```
void glDeleteShader(GLuint sd);
void glDeleteProgram(GLuint pd);
```

Before you run your program, you can validate it against assumed current OpenGL states:

```
void glValidateProgram(GLuint pd);
```

then call `glGetProgram()` to retrieve the result of `glValidateProgram()`, e.g.,:

```
GLint status = 0;
glGetProgramiv(pd, GL_VALIDATE_STATUS,
&status);
if (status == GL_TRUE)
   // program is activated and will execute
```

Integrating GLSL with OpenGL

Two main topics:

1. How do you get your shader codes into your OpenGL program?
   - in this course, all done for you in `shaders.cpp`
2. How do you pass data from your OpenGL program to your shaders?

Setting `uniform` Variables

First you need to retrieve the index of the `uniform` variable associated with the shader program:

```
Glint idx =
   GLint glGetUniformLocation(GLuint pd,
                              const char* varname);

// e.g.,: for uniform variable named "time":
GLint time_idx = glGetUniformLocation(pd, "time");
```

Then you can set the `uniform` variable:

```
glUniform*(Glint idx, TYPE value);

// e.g.,:
void glUniform3f(time_idx, hour, min, sec);
```

Setting `attribute` Variables

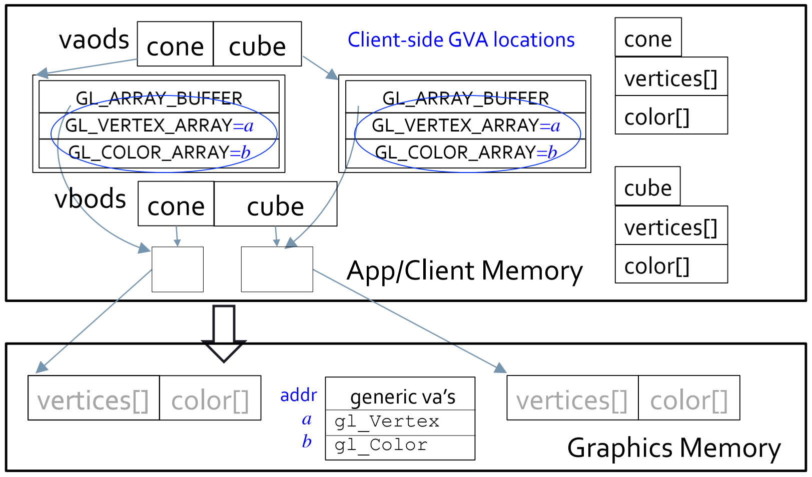
First you need to retrieve the index of the `attribute` variable associated with the shader program:

```
Glint idx =
  GLint glGetAttribLocation(GLuint pd,
                            const char* varname);

// e.g.,: for attribute variable named "color":
GLint color_idx = glGetAttribLocation(pd, "color");
```

Then you can set the `attribute` variable, inside a `glBegin`/`glEnd` block:

```
glVertexAttrib*(Glint idx, TYPE values);

// e.g.,:
void glVertexAttrib4fv(color_idx, nice_color);
```

Generic Vertex Attributes



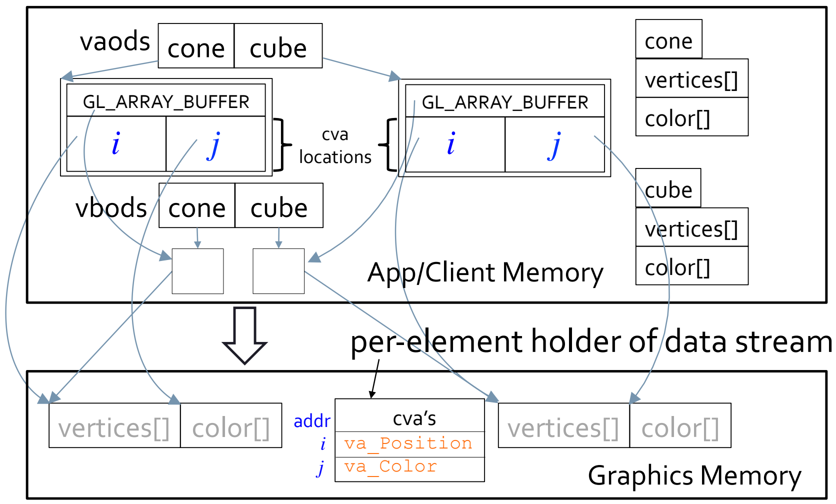
Shader Custom Vertex Attributes

OpenGL 3.1+ and OpenGL ES 2.0+ do not support client-side vertex array and client-side generic vertex attribute locations (i.e., no more `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, etc.)

Instead:

- application uploads vertex data to one or more VBO's (usually directly from .obj or .dds model files; vertex data not defined manually)
- vertex shader declares custom vertex attribute (cva) variables
- application obtains the location of each cva
- application enables each cva and bind it to a VBO: specifying the data format and byte offset from the start of buffer

Vertex Passing in 3.1+ and ES 2.0+



The received way of passing vertices to GPU as of 3.1

Shader Custom Vertex Attributes

Custom vertex attribute (cva) is the variable used by shader to hold each element of the vertex attribute data stream, e.g., the coordinates of a single vertex

App gets the location of cva in shader memory as with any other attribute global variables:

```
int loc = glGetAttribLocation(GLuint pd,
                              GLchar* varname);
```

`//` `pd`: program descriptor, `varname`: cva name

(can use `glBindAttribLocation()` instead,
but binding must be done before call to `glLinkProgram()`)

Custom Vertex Attribute Example

```
attribute vec4 va_Position;
attribute vec4 va_Color;
varying vec4 fa_Color;

void
main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix*va_Position;
  fa_Color = va_Color;
}
```

Application loads, compiles, and links shaders as usual

Then application obtains the cva locations:

```
int vPos = glGetAttribLocation(pd, "va_Position");
int vColor = glGetAttribLocation(pd, "va_Color");
```

Binding CVA to Data Stream

Setup VBO `GL_ARRAY_BUFFER` as usual
(see last lecture and sample code `vao-cva.cpp`)

Enable cva:

```
glEnableVertexAttribArray(vPos);
```

Then bind the cva to theVBO holding the data stream:

```
glVertexAttribPointer(vPos, 2, GL_FLOAT, GL_FALSE, 0,
                      (GLvoid *) 0);  // see next slide
```

Vertex Attribute Stream

Specify stream data format, stride, and byte offset from the start of buffer

```
glVertexAttribPointer(loc, size, type,
  normalized, stride, pointer/byte offset);
```

- `size`: if cva's data type is larger than array element's data type, the extra values default to $(0, 0, 0, 1)$; e.g., `vec4` cva given `vec3f` data means the last component defaults to 1.0

Vertex Attribute Stream

Specify stream data format, stride, and byte offset from the start of buffer

```
glVertexAttribPointer(loc, size, type,
  normalized, stride, pointer/byte offset);
```

Use of `stride` and `offset`:

- prefer all attributes to be specified in one VBO instead of a separate VBO per attribute
- prefer attributes to be interleaved so that various attributes of a single vertex are co-located and can be fetched together
- align each vertex-attribute block with 32-byte cache line

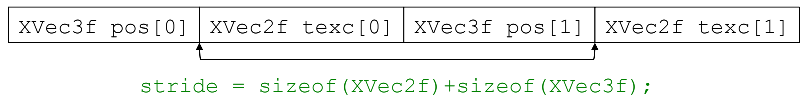
Vertex Attribute Stream

Specify stream data format, stride, and byte offset from the start of buffer

```
glVertexAttribPointer(loc, size, type,
  normalized, stride, pointer/byte offset);
```

- `stride` is the byte offset between the start of an array element and the subsequent data element (for example `stride` $= 0$ if elements are back-to-back):

```
glVertexAttribPointer(texcoords, 2, GL_FLOAT,
      GL_FALSE, sizeof(XVec2f)+sizeof(XVec3f),
      sizeof(XVec3f));
```



Setup and Use of VAOs

```
// Initialize the vertex position and color attributes
// in vertex shader program at handle "spd"
GLuint vert = glGetAttribLocation(spd, "va_Position" );
GLuint color = glGetAttribLocation(spd, "va_Color" );

//set up the two vertex-array objects, assuming vaods generated
glBindVertexArray(vaods[CONE]);
glBindBuffer(GL_ARRAY_BUFFER, vbods[CONE]);
glEnableVertexAttribArray(vert);
glVertexAttribPointer(vert, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(color);
glVertexAttribPointer(color, 3, GL_FLOAT, GL_FALSE, 0, sizeof(cone));

glBindVertexArray(vaods[CUBE]);
glBindBuffer(GL_ARRAY_BUFFER, vbods[CUBE]);
glEnableVertexAttribArray(vert);
glVertexAttribPointer(vert, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(color);
glVertexAttribPointer(color, 3, GL_FLOAT, GL_FALSE, 0, sizeof(cube));
```

VAOs used as previously shown (see `vao-cva.cpp`)

Performance Considerations

In general, minimize:

- Draw calls
- Bind calls for buffer, texture, and program
  - favor all attributes in one VBO over one attribute per VBO
- uniform variable changes
- enable/disable state changes

[Mo]

Common Mistakes (GLSL 1.2)

Shader code has no effect; if there's an error in shader code, OpenGL reverts back to fixed-function pipeline

- make sure variable names used in OpenGL program and in shaders are matched
- built-in variable names has `gl_` prefix
- check vector length:
  - `gl_Normal: vec3`
  - `gl_Vertex: vec4`

OpenGL Extensions

OpenGL extensions provide new rendering features that are not yet part of the OpenGL standard

To keep up with the latest innovations in graphics hardware and rendering algorithms between OpenGL versions

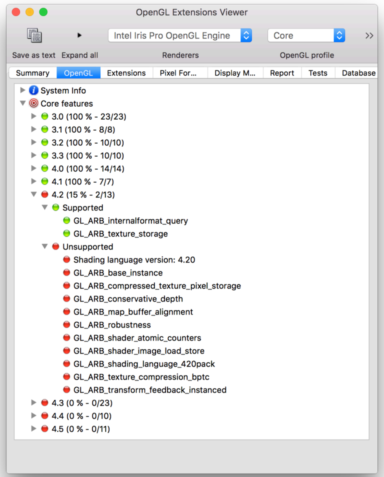
Different kinds of extensions:

- those ratified by the ARB, e.g., `GL_ARB_multitexture();`
- those agreed upon by multiple vendors, e.g., `GL_EXT_abgr();`
- vendor specific ones, e.g., `GL_NV_register_combiners();`

GLview

To see which OpenGL extensions your graphics card and driver support, download Glview (a.k.a. OpenGL Extensions Viewer)

- http://www.realtech-vr.com/glview/
- supports
  - OpenGL 1.1 to 4.5
  - Android, iOS, Mac OS X, Windows
  - AMD, nvidia, and others



Renderers: Intel Iris Pro OpenGL Engine
OpenGL profile: Core

```
▶ System Info
▼ Core features
    ▶ ● 3.0 (100 % - 23/23)
    ▶ ● 3.1 (100 % - 8/8)
    ▶ ● 3.2 (100 % - 10/10)
    ▶ ● 3.3 (100 % - 10/10)
    ▶ ● 4.0 (100 % - 14/14)
    ▶ ● 4.1 (100 % - 7/7)
    ▼ ● 4.2 (15 % - 2/13)
        ▼ ● Supported
            ● GL_ARB_internalformat_query
            ● GL_ARB_texture_storage
        ▼ ● Unsupported
            ● Shading language version: 4.20
            ● GL_ARB_base_instance
            ● GL_ARB_compressed_texture_pixel_storage
            ● GL_ARB_conservative_depth
            ● GL_ARB_map_buffer_alignment
            ● GL_ARB_robustness
            ● GL_ARB_shader_atomic_counters
            ● GL_ARB_shader_image_load_store
            ● GL_ARB_shading_language_420pack
            ● GL_ARB_texture_compression_bptc
            ● GL_ARB_transform_feedback_instanced
    ▶ ● 4.3 (0 % - 0/23)
    ▶ ● 4.4 (0 % - 0/10)
    ▶ ● 4.5 (0 % - 0/11)
```

GL Extension Wrangler (GLEW)

Cross-platform C/C++ extension loading library

- but not needed on Mac OS X

Provides run-time mechanisms for determining which OpenGL extensions are supported

- up to OpenGL 4.5

OpenGL core and extension functionality is exposed in a single header file

- http://glew.sourceforge.net/

GLEW

Include "`glew.h`" before any other GL/GLUT header

Call `glewInit()` right after GL context (GLUT) setup

```
#include <GL/glew.h>
#include <GL/glut.h>
…
glutInit(&argc,argv);
glutCreateWindow("sample");

Glenum err = glewInit();
```

FX Languages

To achieve certain effects, all the shaders must work together

- output of one must match the input of the other
- multiple shader programs may be needed to achieve an effect
- multiple passes may be needed
- there may also be global setting of states

An effects file encapsulates all the relevant info needed to achieve a particular rendering effect

- how to render plastic material
- how to achieve certain shading, e.g., Gooch

Effect languages: HLSL FX, COLLADA FX